Das UNIX Kursbuch

Lutz Wegner

Das UNIX Kursbuch

Eine methodische Einführung in zehn Lektionen

unter Mitarbeit von

**Robert Belting,
Thomas Böntgen,
Wolfgang Brehm,
Wolfgang Frees und
Jens Thamm**

4., korrigierte Auflage Oktober 2004

Prof. Dr. Lutz Wegner
FB 17 Mathematik/Informatik
Universität Kassel
D-34109 Kassel
wegner@db.informatik.uni-kassel.de

Vorwort

Es spricht für die Qualität von UNIX®, wenn dieses Betriebssystem auch nach fast 30 Jahren noch die bevorzugte Plattform für alle modernen, professionellen Softwareentwicklungen ist. Das Geheimnis des Erfolgs liegt dabei in der Beschränkung auf wenige, logisch konsistente Grundprinzipien. Dazu gehören die einheitlichen System- und Anwenderprogramme, die Gleichbehandlung von Dateien und Geräten, die Modularität der Kommandos, die austauschbare Benutzerschnittstelle und das offengelegte Prozeßkonzept.

Auch spätere Generationen von Entwicklern, die Komponenten wie die GNU-Umgebung, X-Windows oder in den letzten Jahren die PC-Versionen LINUX und FreeBSD beigetragen haben, blieben dieser gerne als UNIX-Philosophie bezeichneten Vorgehensweise treu. Daneben hat UNIX auch einen eigenen, etwas spröden Stil entwickelt, der sich wohltuend von den Aufgeregtheiten der Industrie abhebt. Das ist sicherlich den Gründungsvätern und ihrem Witz zu verdanken.

Das vorliegende Buch versucht, diese Philosophie herauszuarbeiten und dem UNIX-Stil gerecht zu werden. Es beruht auf Vorlesungen und Weiterbildungskursen zu UNIX, die ich und meine Mitarbeiter über die Jahre abgehalten haben. Hauptziel ist der sichere Umgang mit UNIX. Dazu gehört einerseits das Kennenlernen der wichtigsten Kommandos und des hierarchischen Dateisystems, dargestellt in lebendigen Szenarien aus dem UNIX-Alltag, andererseits das Verständnis der grundlegenden Konzepte, wie z. B. die Prozeßumschaltung oder die getrennten Sicherheitsbereiche.

Der Stoff ist in sich abgeschlossen, vorausgesetzt werden nur elementare DV-Kenntnisse. Etwas Programmiererfahrung erleichtert das Verständnis der späteren Lektionen, die den Schwerpunkt auf die Bourne-Shell legen. Zahlreiche Abbildungen, Zusammenfassungen und über einhundert Kontrollfragen mit Lösungen sollen den Einstieg erleichtern und machen den Text auch für das Selbststudium geeignet.

Ferner sind in den meisten Abschnitten der zehn Lektionen Verweise auf weitere Themen eingestreut, die an doppelten Linien davor und danach erkennbar sind. Diese Kommentare, die nicht den Anspruch erheben, einen umfassenden Überblick über alle Werkzeuge, Oberflächen und Entwicklungsrichtungen zu geben, können beim ersten Lesen überschlagen werden.

Idealerweise wird der Leser Zugang zu einem UNIX-Rechner haben. Beim Üben wird er feststellen, daß die Bildschirmausgaben des Buchs manchmal von den am Rechner beobachteten abweichen. Die Abweichungen sind aber nicht prinzipieller Natur. Vielmehr liegen sie zum einen an den unterschiedlichen „UNIX-Dialekten“, zum anderen an Kürzungen in der Ausgabe, die zur Straffung der Abbildungstexte vorgenommen wurden.

Alle Beispiele wurden an mehreren unabhängigen Systemen getestet. Genauso sind die Kontrollfragen, die bewußt nicht immer einfach, aber möglichst eindeutig gestellt sind, durch einige Jahrgänge von Studenten gegangen.

An der Gestaltung dieses UNIX-Kurses haben in Fulda Robert Belting, Thomas Böntgen und Wolfgang Brehm, sowie in Kassel Wolfgang Frees und Jens Thamm mitgearbeitet. Dem B^3-Team in Fulda ist dabei die äußerst sorgfältige Gestaltung von Text, Graphiken und Animation einer frühen, elektronischen Version für computergestützten Hypermedia-Unterricht zu verdanken. Dieses Projekt ging auf eine Anregung von Prof. Maurer, Graz, zurück.

Wolfgang Frees hat den Rohtext für ein Skript zum Kurs mit einem DTP-System bearbeitet und Graphiken hierfür entworfen. Den jetzt vorliegenden Text und die Graphiken, die ich mit FrameMaker® auf einer RS/6000 Workstation unter AIX gestaltet habe, hat Jens Thamm durchgesehen und als hervorragender UNIX-Kenner wesentlich ergänzt. Nicht vergessen möchte ich auch Burkhardt Fischer, der unsere heterogene UNIX-Umgebung eingerichtet hat und sie über alle Release-Wechsel zuverlässig am Laufen hält, sowie Reda Khalifa und Sven Thelemann, die mich in Sachen UNIX und FrameMaker unterstützt haben und mich auf viele interessante Details aufmerksam machen konnten. Jochen Ruhland hat dank gründlicher zweiter Durchsicht weitere verbliebene Druckfehler und falsche Angaben herausgefunden und aufgelistet.

Allen hier genannten Personen gilt mein ganz herzlicher Dank. Trotzdem gehen alle Fehler und Mißverständnisse zu meinen Lasten. Anregungen und Hinweise werden gerne entgegen genommen.

Kassel, im Oktober 1997

Lutz Wegner

Inhaltsverzeichnis

LEKTION 1: Einführung

1.1 Geschichte und Geschichten

❖ In diesem Abschnitt stellen wir einige historische Fakten und „Histörchen“ aus der Vergangenheit von UNIX vor und erläutern den gegenwärtigen Stand.

UNIX - ein modernes Märchen?

Die Anzahl der UNIX-Installationen ist auf 10 gewachsen, und mehr werden erwartet.

UNIX Programmierhandbuch
2. Auflage Juni 1972

Es war einmal eine kleine, wenig benutzte DEC PDP-7 und ein unzufriedener Ken Thompson. Das Jahr ist 1969, der Ort die Bell Laboratories. Thompson beschließt, ein kleines Betriebssystem zu schreiben. Dennis Ritchie stößt rasch dazu. Umgeschrieben in Assembler für die PDP-11 läuft es im Februar 1971.

In Anspielung auf das Vorläufer-Betriebssystem Multics erhält es den Namen UNIX.

Der Rest ist Geschichte:

- 1973 entwickeln Ken Thompson und Dennis Ritchie die Programmiersprache C und schreiben UNIX in C um.
- 1974 erscheint ihr berühmter Übersichtsartikel „The UNIX Time-Sharing System“ in den Communications of the ACM [10] und lenkt das Interesse vieler Forschungseinrichtungen auf UNIX.
- 1978 existieren etwa 600 Installationen und Western Electric beginnt die Vermarktung.
- 1979 entsteht nach dem vierten Umschreiben die Version 7. Sie ist Ziehmutter vieler Abkömmlinge, unter anderem der BSD Systeme der University of California at Berkeley.
- 1982 geben die AT&T Bell Laboratories **System III** frei, die Portierungen auf Motorola 68000 Rechnern nehmen zu.
- 1984 folgt **System V** bzw. **4.2 BSD**. Diese Systeme bilden die Grundlage für die heutigen Releases (vgl. Gulbins/Obermayer [6] für eine sehr gründliche Einführung).
- 1986 erscheint der IEEE Standard 1003.1, bekannt als POSIX Standard, der einen Kern von UNIX-Funktionalität normiert.
- ab 1991 entsteht die Open Software Foundation (OSF) zunächst als Antwort auf Versuche von AT&T und Sun Microsystems, die Entwicklung von UNIX wieder zu monopolisieren. Die Mitglieder IBM, Hewlett-Packard, DEC, Bull, Siemens, u. a. bringen **OSF/1** auf der Basis der IBM-Variante **AIX** auf den Markt.
- ab 1985 wird an der Carnegie-Mellon-University ein neuer UNIX-Kern namens **MACH** entwickelt, dessen kommerzielle Variante im Betriebssystem NeXTstep eingesetzt wird und das auch in zukünftige Entwicklungen der OSF einfließen soll.
- Die Gründerväter von UNIX arbeiten derweil in den Bell Labs an **Plan9**, das keine Weiterentwicklung von UNIX ist, sondern einen kompletten Neuentwurf darstellt, der inzwischen auch schon wieder 10 Jahre alt ist.
- ab 1992 ist **Linux** für Rechner mit Intel x86 Prozessor verfügbar, das von dem Finnen Linus B. Torvalds geschaffen wurde und public domain ist. Es enthält viele der für UNIX frei verfügbaren Komponenten, wie z. B. das **X11** Fenstersystem und die **GNU**-Produkte der Free Software Foundation.
- Auch gut geeignet für die heutige Generation preiswerter, aber leistungsfähiger PCs ist **FreeBSD** (Quelle: `www.de.freebsd.org`). Mit Linux und FreeBSD kehrt UNIX in gewisser Weise wieder zu seinen Wurzeln der freien Verfügbarkeit zurück.

Kein Märchen ohne Happy End!

- 1983 erhalten Dennis Ritchie und Ken Thompson für die Entwicklung und Implemen-

tierung des UNIX Betriebssystems die höchste Auszeichnung in der Informatik, den ACM A.M. Turing Award.

Aus dem Eigengewächs zweier talentierter Programmierer entsteht der Stammbaum für den Betriebssystemstandard der achtziger Jahre.

Weil es sein muß

UNIX ist eingetragenes Warenzeichen mit exklusiver Lizenzvergabe durch die X/Open Company Limited ● DEC, PDP und VAX sind Warenzeichen der Digital Equipment Corporation ● XENIX ist Warenzeichen der Microsoft Corporation ● SINIX ist Warenzeichen der Siemens AG ● POSIX ist Warenzeichen der IEEE ● AIX ist Warenzeichen der International Business Maschines Corporation ● OSF ist eingetragenes Warenzeichen der Open Software Foundation ● SunOS und Solaris sind eingetragene Warenzeichen der Sun Microsystems Inc. ● HP-UX ist Warenzeichen der Hewlett Packard Company

● SIMULIX ist Wahrzeichen dieses Kurses.

Weil es sein könnte, daß wir Ihr System nicht genannt haben: Verzeihung, wir haben nur eine Auswahl vorgestellt.

Zusammenfassung

❖ Wir haben die ungewöhnliche Entwicklungsgeschichte des Betriebssystems UNIX und die verwirrende Namensgebung der Versionen kennengelernt.

Frage 1

Der Erfolg von UNIX beruht auf der Effizienz seiner Implementierung im Assembler der Zielmaschine. Ist diese Behauptung richtig oder falsch?

Frage 2

UNIX war eine Auftragsarbeit der AT&T Bell Laboratories. Ist diese Behauptung richtig oder falsch?

Frage 3

Wenn es UNIX in so vielen Versionen gibt, warum ist es dann trotzdem de facto ein Standard?

❏ Trotz der unterschiedlichen Benutzeroberflächen sind der Kern, das Dateisystem und das Prozeßkonzept einheitlich.

❏ Trotz unterschiedlicher interner Struktur sind alle Benutzeroberflächen einheitlich.

1.2 Am Terminal

❖ In diesem Abschnitt simulieren wir die erste Begegnung mit UNIX an einem Bildschirmterminal. Der Dialog wird von einem fiktiven Benutzer geführt.

Gestatten, mein Name ist Professor Fix. Begleiten Sie mich auf meinen ersten Schritten mit unserem neuen UNIX Rechner. Ich habe uns ein Terminal besorgt, schließe es eben schnell an und schalte es nun ein:

Professor Fix gibt seinen Benutzernamen (login-Name) ein.

```
login: fix
```

Die Eingabe schließt er mit der Eingabetaste 'RETURN' ab.

```
login: fix
password:
```

Das System verlangt ein Paßwort. Professor Fix gibt sein Paßwort ein.

```
login: fix
password:........
```

Es bleibt unsichtbar und er drückt wieder die Eingabetaste.

```
login: fix
password:........
-----------------------------------------
! MITTEILUNGEN DES SYSTEMVERWALTERS:      !
! Heute keine Wartung                     !
! ein weiterer Teilnehmer: Prof. Fix      !
! Herzlich Willkommen                     !
-----------------------------------------
$
```

Nach einigen Systembotschaften erscheint das Dollarzeichen am Terminal. Damit wird die Empfangsbereitschaft des Systems für Eingaben angezeigt. Wir sind im System!!!

Verschaffen wir uns nun einen Eindruck vom Umgang mit UNIX. Nach einer alten Informatikweisheit sollten einfache Dinge auch einfach gehen. Wie wär's daher mit der Ausgabe von „Hallo Leute!“ am Bildschirm?

```
$echo Hallo Leute!
```

Auch diese Eingabe wird, wie alle Benutzereingaben, wieder mit der Eingabetaste 'RETURN' abgeschlossen. Darauf werden wir zukünftig nicht mehr besonders hinweisen.

```
Hallo Leute!
$
```

Weil es so leicht war, gleich noch ein Beispiel.

```
$Echo Aller Anfang war leicht!
Echo: not found
$
```

Das ging daneben! Das große `E` von `Echo` war Schuld, denn UNIX unterscheidet Groß- und Kleinschreibung. Mit wenigen Ausnahmen wird alles klein geschrieben, auch login-Namen und Kommandos. Versuchen wir es also noch einmal.

```
$echo Aller Anfang war leicht!
Aller Anfang war leicht!
$
```

Bevor wir uns wieder abmelden, wollen wir schnell noch unser Paßwort ändern:

```
$passwd
Old password:........
New password:......
Retype new password:......
$
```

Nach der Eingabe des Kommandos `passwd` werden wir aufgefordert

- unser altes Paßwort einzugeben,
- unser neues Paßwort einzugeben und
- unser neues Paßwort zu wiederholen.

Alle Paßworteingaben bleiben unsichtbar. Warum muß man wohl zuerst das alte Paßwort und dann zweimal das neue Paßwort eingeben?

Ohne das alte Paßwort könnte bei unserer Abwesenheit vom angemeldeten Terminal jemand das Paßwort ändern, und wir wären bei der nächsten Sitzungseröffnung ausgesperrt.

Ohne zweimalige Eingabe des neuen Paßwortes könnte uns ein (unsichtbarer) Tippfehler entgehen — wir hätten uns selbst ausgesperrt.

Nur der Besitzer des Paßwortes und der Systemverwalter können das Paßwort ändern, letzterer z. B. durch `passwd fix`, wenn Herr Professor wieder einmal sein Paßwort vergessen hat. Der Systemverwalter heißt in UNIX super-user. Ihn müssen wir auch vor der ersten Terminalsitzung aufsuchen, damit er uns unter unserem login-Namen im System einträgt.

Und wie meldet man sich ab?

Vielleicht mit `logoff`, `logout`, `goodbye` oder ähnlichem, wenn ein barmherziger Mensch eines dieser Kommandos in Ihrem System installiert hat. Meist aber mit `exit` oder mit ‘CTRL-d’ (gleichzeitiges Drücken der ‘CTRL’- und der ‘d’-Taste). Letztgenannte Tastenkombination erzeugt das ASCII-Zeichen EOT (end of transmission, oktal 004), das in UNIX generell das Eingabe- bzw. Dateiende signalisiert.

```
$echo Jetzt mache ich Schluss.
```

```
Jetzt mache ich Schluss.
$CTRL-d
```

Damit ist das Terminal für die nächste Sitzung frei.

Zusammenfassung

❖ Wir haben die Sitzungseröffnung (login), ein einfaches Kommando (`echo`), das Ändern des Paßwortes (`passwd`), die Rolle des Systemverwalters und die Sitzungsbeendigung kennengelernt.

Frage 4

Der login-Name ist

❏ der geheimzuhaltende Zutrittsschlüssel zum System,

❏ eine vom Systemverwalter gewählte Nummer, auch user-id genannt,

❏ ein freigewählter Benutzername.

Frage 5

Welchen einprägsamen Namen hat der mit besonderen Rechten ausgestattete Systemverwalter in UNIX?

Frage 6

Wie lautet die magische Tastenkombination zum Abmelden (Beenden einer Terminalsitzung)?

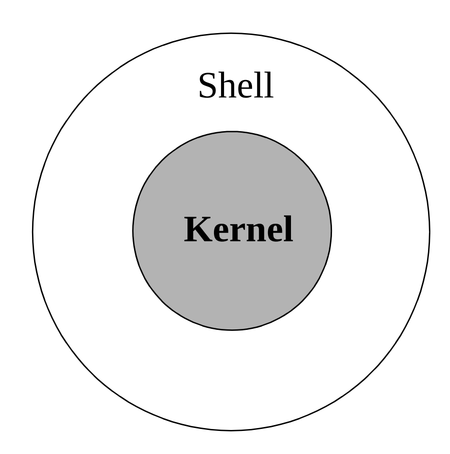
1.3 Auf mein Kommando

❖ In diesem Abschnitt stellen wir das Grundkonzept des Kommandointerpreters und der Kommandos vor. Die Konzepte werden in den weiteren Lektionen vertieft.

Jedes Betriebssystem hat mindestens zwei Komponenten:

- Einen *Betriebssystemkern* (UNIX: *Kernel*) zur Verwaltung von System- und Benutzerprozessen (Tasks).
- Einen *Kommandointerpreter* (UNIX: *Shell*) zur Entschlüsselung und gegebenenfalls Ausführung von Benutzerwünschen.

Der Name Shell (engl. Muschel, Schale) deutet an, daß der Kommandointerpreter in UNIX den Kernel umgibt und ihn vor dem direkten Benutzerzugriff schützt.



Die Shell führt in der Regel selbst keine Kommandos aus. Sie interpretiert nur die Eingabezeile und gibt die Argumente der erkannten Kommandos, eventuell modifiziert, an Programme weiter, welche die Kommandos realisieren. Drei solcher Kommandos kennen wir schon: `login`, `echo` und `passwd`.

In `echo Hallo Leute!` ist `echo` also der Kommandoname, `Hallo` und `Leute!` sind die Argumente. Argumente werden durch mindestens eine Leerstelle vom Namen und untereinander getrennt. Damit hat

```
echo Hallo Leute!
```

zwei Argumente

```
passwd fix
```

eines und

```
passwd
```

null Argumente.

Gleichzeitig ist jeder Kommandoname auch Bezeichner der Datei, welche den ausführbaren Programmcode für dieses Kommando enthält. Wo diese Dateien genau stehen, also z. B. die Dateien `echo`, `passwd` und `login`, erläutern wir in einer späteren Lektion.

Die Verarbeitung der Eingabe `echo Hallo Leute!` zeigen wir schematisch im nächsten Bild.

Halten wir fest: Betriebssystemkern, Kommandointerpreter und Kommandos bilden *keinen monolithischen Block*. Jedes einzelne Kommando und auch die Shell sind einzelne, meist in der Programmiersprache C erstellte Programme. Sie sind damit leicht austauschbar, erweiterbar und überschaubar.

Die Austauschbarkeit macht folglich auch nicht vor dem Kommandointerpreter halt.

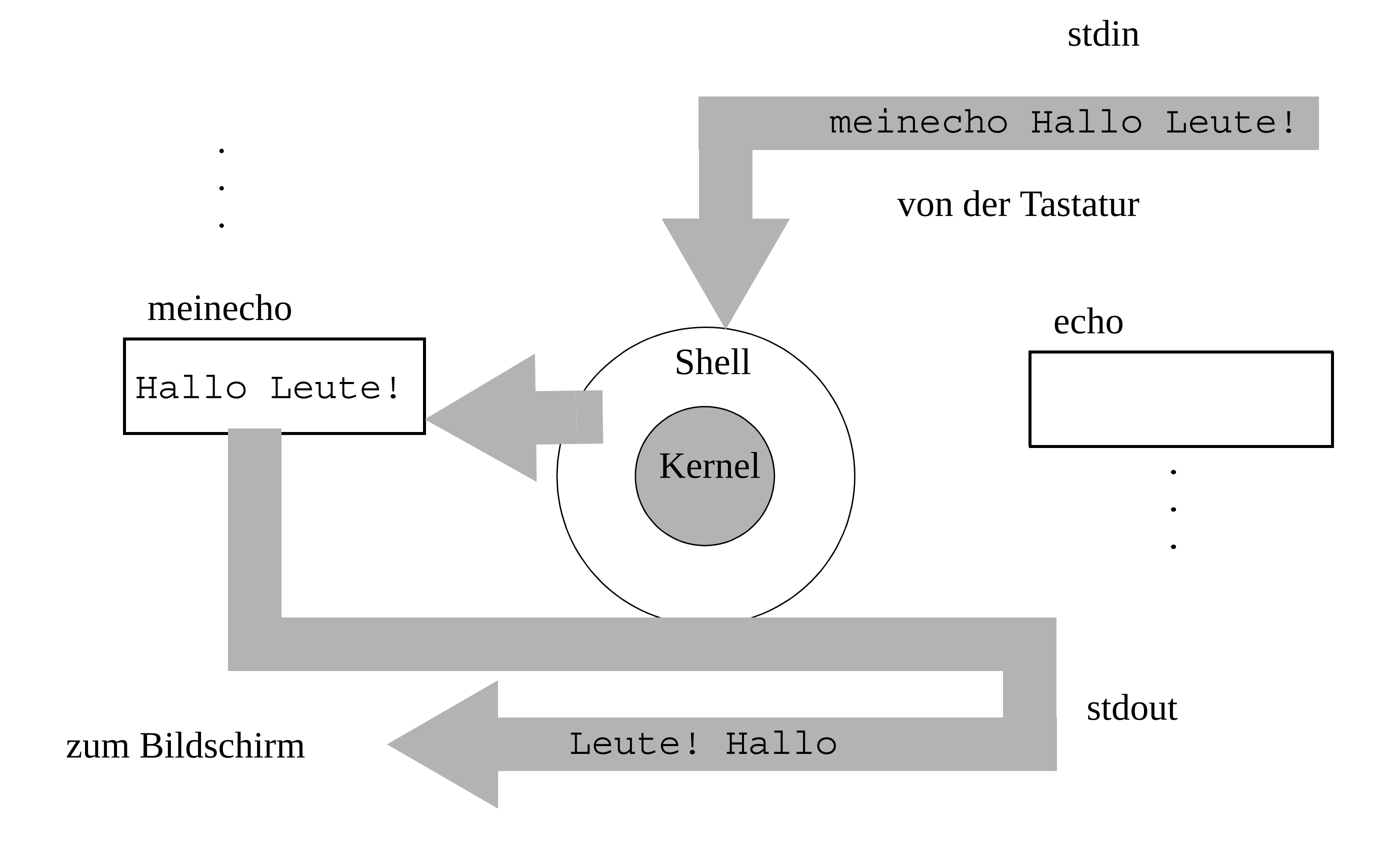
Die hier besprochene sogenannte *Bourne-Shell* mit dem Programmnamen `sh` kann von jedem Benutzer jederzeit gegen eine Shell mit C-Syntax, die sogenannte *C-Shell* mit dem Programmnamen `csh`, gegen die *Korn-Shell* (`ksh`), die eine Erweiterung der Bourne-

stdin
echo Hallo Leute!
login
von der Tastatur
passwd
Shell
Kernel
echo
Hallo Leute!
stdout
zum Bildschirm
Hallo Leute!

Shell ist oder eine andere verfügbare Shell, z. B. die *Bourne Again Shell* (`bash`), getauscht werden.

Genauso können neue, eigene Kommandos eingeführt werden, z. B. ein wortweise spiegelndes Echo namens `meinecho`.

Auch vorhandene Kommandos können ausgetauscht werden mit Wirkung



- nur für einen Benutzer (durch ihn selbst),
- für alle Benutzer (durch den super-user),
- für eine UNIX-Systemfamilie, z. B. zur Unterstützung einer Kommunikationsumgebung (durch einen Systementwickler).

Welche einfachen Konventionen für das Zusammenspiel der Programme und Daten dabei zu beachten sind, erläutern die folgenden Lektionen.

Zeilen-, seitenorientiert oder graphisch?

Eine (Kommando-)Zeile eingegeben, ein paar Zeilen zurückbekommen — ist das etwa die UNIX Benutzungsschnittstelle? Wo bleiben die Fenster, die Icons, die Menüs, die Interaktion mit der Maus?

Die schlechte Nachricht ist, daß die UNIX-Oberfläche im Prinzip *zeilenorientiert* ist. Daneben gibt es Kommandos und Werkzeuge, z. B. der weiter unten vorgestellte Editor `vi` oder das Postbearbeitungswerkzeug `elm`, die *seitenorientiert* sind, d. h. bei denen man wenigstens mit den Cursortasten hinauf- und hinunterfahren, mehrere Zeilen markieren und blättern kann. Sowohl zeilen- als auch seitenorientierte Interaktion kann auf textorientierten (ASCII) Terminals ablaufen, deren klassischer Vertreter das VT100 Terminal war, weshalb dieser Modus oft VT100-Emulation heißt.

Die gute Nachricht ist, daß es für UNIX durchaus *graphische Benutzeroberflächen* gibt. Grundlage für diese Oberflächen ist meist das sehr fortschrittliche, verteilte X-Windows, dessen gegenwärtige Version einfach als X11 bekannt ist. X wurde zunächst ab etwa 1985 vom MIT als public domain Software veröffentlicht, ab 1989 von der ISO als Standard festgeschrieben. Als weiterführende Literatur sei auf Klingert [8] verwiesen, der eine schöne Übersicht über Konzepte graphischer Fenstersysteme gibt.

Auf dem mit recht elementaren Funktionen ausgestatteten X11 bauen eine Reihe weiterer, komfortablerer Fenstersysteme auf, z. B. das von der OSF forcierte *Motif*, das auch unter Linux, allerdings nicht kostenlos, verfügbar ist. Auch andere graphische Anwendungen setzen in der UNIX-Version gerne X11 als Basis voraus, etwa die erweiterbare Schnittstellensprache und Werkzeugbox *Tcl/Tk* von John Ousterhout [9], der Web-Browser *Netscape* und die UNIX-Version von *Java*.

Die vielleicht beste Nachricht ist, daß UNIX die graphikorientierten Oberflächen nicht zum Prinzip erhebt, sondern als eine komfortable Erweiterung der Möglichkeiten betrachtet. Die meisten Werkzeuge laufen deshalb auch in einer bescheideneren, textorientierten Umgebung und wo nicht, gibt es textorientierte Ausweichwerkzeuge, so etwa `lynx` als Ersatz für Netscape.

Ein positives Beispiel ist das frei verfügbare Werkzeug `xman` zum Anschauen des Online-Handbuchs (der Manual Pages, vgl. Abschnitt 2.1 unten) unter X11. Dieses bietet eine ansprechende interaktive Hülle um die Handbuchseiten, die traditionell unter UNIX vor-

formatiert für eine Druckausgabe abgelegt sind. Diese Vorformatierung berücksichtigt auch die Ausgabe auf einem „dummen“ ASCII-Terminal.

Wird dies nicht berücksichtigt, etwa in dem interaktiven, hypertextartigen Info-Explorer unter AIX, der die traditionellen Manual Pages ersetzt, ist die Navigation in einer graphischen Umgebung zwar recht nett, die reine Textausgabe kann dagegen nicht befriedigen.

Wer sich schon einmal per Modem über die nicht besonders schnellen Leitungen des Telefonnetzes bei einem entfernten Rechner eingeloggt hat, z. B. um nur schnell in der eigenen Mailbox nach eingelaufenen Nachrichten zu sehen, weiß die karge Schlichtheit einer VT100 Emulation zu schätzen. Wer wirft schon gerne sein Geld zum Fenster hinaus?

Zusammenfassung

❖ Wir haben die Rolle des Kommandointerpreters (der Shell), den Aufbau eines Kommandos, das Zusammenspiel von Shell und Kommandos, die Realisierung von Kommandos als ausführbare Programmdateien und das Prinzip der Austauschbarkeit kennengelernt.

Frage 7

Wieviele Argumente hat das Kommando:

```
echo Hallo Ihr da !
```

Frage 8

Die am meisten verbreitete Shell (Name: `sh`) ist

❏ die Bourne-Shell,

❏ die C-Shell,

❏ die visual-Shell.

Frage 9

Welche der folgenden drei Kommandozeilen könnte eine (Fehler-) Meldung des Systems bewirken?

❏ `echo echo`

❏ `echo passwd`

❏ `passwd echo`

LEKTION 2: Ein- und Ausgabeumlenkung

2.1 Einfach weiter

❖ In diesem Abschnitt stellen wir weitere einfache Kommandos vor, die bereits den typischen Umgang mit UNIX verdeutlichen.

Die Liste der Kommandos in UNIX enthält über 300 Einträge. Davon sind rund 100 Sprachübersetzer, Textaufbereitungs- und Softwareentwicklungswerkzeuge. Unmöglich, dies alles im Kopf zu behalten. Auskunft liefert das *on-line Benutzerhandbuch*.

```
$man echo
ECHO(1)                                        ECHO(1)
            UNIX Programmer's Manual
NAME
    echo - echo arguments
SYNOPSIS
    echo [-n] [arg] ...
DESCRIPTION
    Echo writes its arguments usw.
$
```

Dabei ist `man` der Kommandoname für das on-line Benutzerhandbuch, `echo` das Argument, d. h. der Name des Kommandos, über das eine Auskunft gewünscht wird.

Die Erläuterungen haben eine feste Form mit wenigstens:

- **NAME** Mnemonische- und Vollbezeichnung.
- **SYNOPSIS** Syntaxbeschreibung mit eckigen Klammern [ ] für optionale Teile und Punkten ... für Wiederholungen.
- **DESCRIPTION** Beschreibung der Funktion und der zulässigen Optionen.

Was liefert wohl `man man`?

```
$man man
MAN (1)                                          MAN (2)
             UNIX Programmers's Manual
NAME
      man - print sections of this manual
SYNOPSIS
      man [option ...] [chapter] title ...
DESCRIPTION
      Man locates and prints usw.
$
```

Natürlich die Beschreibung des `man` Kommandos! Weitere Beispiele für einfache Kommandos:

```
$who
peter tty0 Dec 13 08:03
gabi tty2 Dec 13 10:21
jr tty3 Dec 13 08:40
fix tty7 Dec 13 09:16
```

`who` gibt an, *wer* (login-Name) *wo* (**tty** steht für **tele**ty**pe**, d. h. tty7 ist Terminal Nr. 7) *seit wann* (Datum und Uhrzeit) angemeldet ist.

```
$date
Fri Dec 13 10:39:05 GMT 1996
$
```

`date` gibt die gegenwärtige Systemzeit aus bzw. erlaubt dem super-user das Setzen der Zeit.

Professor Fix sendet nun einen elektronischen Brief an den Teilnehmer `jr`.

```
$mail jr
Subject: Erste Schritte mit UNIX
Lieber Kollege !

Danke für den Hinweis auf UNIX. Ein cleveres, wenn auch etwas
sprachfaules System.
```

```
Gruss Fix
Cc:
$
```

Die Texteingabe innerhalb des Kommandos `mail` wird mit ‘CTRL-d’ abgeschlossen. Dabei muß der Empfänger nicht angemeldet sein. Über angekommene Post wird jeder Teilnehmer nach dem Anmelden (login) vom System informiert. In der Zeile `Subject:` kann Fix einen *Betreff-Vermerk* eingeben, der den Empfänger über den Inhalt informiert. In der Cc-Zeile kann der Absender Empfänger weiterer Kopien (*cc* steht im Englischen für *carbon copy*, d. h. Durchschlag) benennen.

Meist stehen in UNIX komfortablere Mail-Werkzeuge zur Verfügung, z. B. `elm` (seitenorientiert) oder ein graphisches System wie `xmail`.

Zusammenfassung

❖ Wir haben die Kommandos `man` und das on-line Manual, `who` zur Auflistung der Teilnehmer, `date` zur Ausgabe der Zeit und `mail` zum Versenden von Briefen kennengelernt.

Frage 1

Die Ausgabe des `who` Kommandos benutzt als Terminalnamen eine Abkürzung mit drei Buchstaben. Wie lautet diese Abkürzung?

Frage 2

Professor Fix gibt das Kommando `mail fix` und einen Brieftext ein. Dadurch wird

❏ ein Brief an den Teilnehmer fix geschickt,

❏ ein Systemfehler hervorgerufen,

❏ ein Brief an alle Teilnehmer geschickt.

Frage 3

Das Dateiendezeichen ‘CTRL-d’ beendet die Eingabe zum mail-Kommando. Es wird aber auch zum Abmelden vom System benutzt. Welche der folgenden Aussagen gilt?

❏ Das ist eine der typischen UNIX-Ungereimtheiten.

❏ Das ist innerhalb der UNIX-Philosophie folgerichtig.

2.2 Mit ed an der Arbeit

❖ In diesem Abschnitt wird ein kleines Arbeitsbeispiel behandelt. Wir verwenden die Editoren `ed` und `vi`, das `mail` Kommando und die Eingabeumlenkung.

Professor Fix sitzt wieder am Terminal und absolviert die uns nun schon bekannte login-Prozedur.

```
SIMULIX, das freundliche Übungssystem
login: fix
Password:......
```

Er gibt sein Paßwort ein und ...

```
------------------------------------------
!   MITTEILUNGEN DES SYSTEMVERWALTERS    !
! - ein weiterer Teilnehmer: Prof. Fix.  !
! Herzlich willkommen                    !
------------------------------------------
You have mail.
$
```

... er hat Post. Er schaut nach, was gekommen ist.

```
$mail
"/usr/spool/mail/fix": 1 messages 1 new
>N   1 jr Fri Dec 13 15:10 GMT 1996 "Engpass Terminals"
>N   2 freundl Fri Dec 13 13:27 GMT 1996 "UNIX Klausur"
& 1
Message 1:
From jr Fri Dec 13 15:10 GMT 1996
Date: Fri, Dec 13 15:10 GMT 1996
From: jr (J. Richtig)
To: fix
Subject: Engpass Terminals

Schoen, dass UNIX Ihnen Spass macht. Bei der naechsten FBR-Sitzung
Engpass bei den Terminals ansprechen!
Mit freundlichem Gruss

J. Richtig
&
```

Der Teilnehmer `jr` hat geantwortet. Mit `mail` ohne Argumente schaut man sich die Post an. Das Fragezeichen, bzw. Und-Zeichen (&) auf manchen Systemen, fordert eine Reaktion. Mögliche Eingaben (und ihre Auswirkung auf die Botschaft) sind:

- **d** (**d**elete) aus dem Postfach entfernen,
- **s datei** (**s**ave) retten in eine Datei und aus dem Postfach entfernen,
- **p** (re**p**rint) Brief nochmals ansehen,
- **q** (**q**uit) Postdurchsicht beenden,
- ‘CTRL-d’ gleiche Wirkung wie q,

- **m name** (**m**ail on) Post weitergeben,
- ’RETURN’ weiter, Botschaft bleibt im Postfach.

Die AIX-Version des Mail-Kommandos - Ausgabe für ? am Mail-Prompt

Control Commands:

q	Quit - apply mailbox commands entered this session.
x	Quit - restore mailbox to original state.
! <cmd>	Start a shell, run <cmd>, and return to mailbox.
cd [<dir>]	Change directory to <dir> or $HOME.

Display Commands:

t [<msg_list>]	Display messages in <msg_list> or current message.
n	Display next message.
f [<msg_list>]	Display headings of messages.
h [<num>]	Display headings of group containing message <num>.

Message Handling:

e [<num>]	Edit message <num> (default editor is e).
d [<msg_list>]	Delete messages in <msg_list> or current message.
u [<msg_list>]	Recall deleted messages.
s [<msg_list>] <file>	Append messages (with headings) to <file>.
w [<msg_list>] <file>	Append messages (text only) to <file>.
pre [<msg_list>]	Keep messages in system mailbox.

Creating New Mail:

m <addrlist>	Create/send new message to addresses in <addrlist>.
r [<msg_list>]	Send reply to senders and recipients of messages.
R [<msg_list>]	Send reply only to senders of messages.
a	Display list of aliases and their addresses.

==================== Mailbox Commands ========================

Professor Fix entfernt den Brief durch die Eingabe **d**. Aber wie Sie sehen,

```
& d
& 2
Message 2:
From freundl Fri Dec 13 13:27 GMT 1996
Date: Fri, Dec 13 13:27 GMT 1996
From:freundl (Freundlich)
```

```
To: fix
Subject: UNIX Klausur

Einige Studenten haben wegen des Ter-
mins zur UNIX-Klausur nachgefragt.
gez. Freundlich
&
```

ist eine weitere Botschaft von der Sekretärin da.

Professor Fix möchte eine Antwort formulieren. Anstatt über `mail` die Antwort direkt einzugeben ist es besser, erst eine Datei anzulegen und dort die Klausurankündigung zu formulieren. Professor Fix verwendet dazu den archaisch einfachen, zeilenorientierten Editor `ed`. Eine Alternative wäre der seitenorientierte `vi` (**v**isual **i**mage), der `ed` beeinhaltet und zu den Berkeley-Erweiterungen von UNIX gehört, oder `emacs`, ein Editor aus der GNU-Softwarefamilie.

Ein recht einsilbiger Dialog entsteht. Zunächst wird `mail` mit `q` (**q**uit) beendet. Dann ruft Professor Fix den Editor `ed` mit dem Argument `ankuend` (Dateiname) auf.

```
& q
$ed ankuend
?ankuend
```

Der Editor teilt mit, daß die Datei `ankuend` nicht existiert und neu angelegt wird.

Mit dem Editor-Kommando `a` (**a**ppend) geht Professor Fix nun in den *Anfügemodus* und gibt seinen Text, der in die Datei geschrieben werden soll, ein.

```
a
Die UNIX-Klausur findet am Freitag,
dem 14. Februar 1997 um 14 Uhr im Raum
1409 statt.
Hilfsmittel sind nicht zugelassen.
            gez. Fix 13/12/1996
.
w
143
q
$
```

Der Punkt, als erstes und einziges Zeichen einer Eingabezeile, führt zurück in den *Kommandomodus*. Das Editor-Kommando `w` (**w**rite) sorgt dafür, daß der eingegebene Text vom Bildschirm (Hauptspeicher) in die Datei übertragen wird.

Der Editor antwortet mit der Anzahl geschriebener Zeichen inkl. der Zeichen NL (*newline,* oktal 012) für „neue Zeile“. Professor Fix beendet das Editieren mit dem Editor-Kommando `q` (**q**uit).

Durch nochmaligen Aufruf von `ed` mit dem Argument `ankuend` könnte Professor Fix die Datei erneut bearbeiten, z. B. weil er den Termin ändern möchte. Üblicherweise wird er auch ab sofort einen etwas vernünftigeren Editor, nämlich `vi`, für kurze Editieraufgaben verwenden, dessen wichtigste Kommandos im folgenden kurz aufgeführt werden.

Der Editor vi

Mit `vi` *Name* wird eine existierende Datei *Name* zum Editieren aufgerufen, bzw. eine neue leere Datei angelegt. Der Editor befindet sich im *Kommandomodus*.

Im Kommandomodus kann man mit den Cursortasten navigieren. Einzelne Zeichen unter dem Cursor lassen sich mit `x` löschen (cross out), mit `dd` wird die gegenwärtige Zeile entfernt.

Zum Einfügen oder Anfügen geht man mit `i`, `a` oder `A` (anfügen am Ende der Zeile) in den *Eingabemodus* und kann jetzt Texte schreiben. Kleinere Tippfehler lassen sich jetzt mit der Rücksetztaste (backspace) beheben, Cursortasten sind allerdings nicht verfügbar. Mit der Escape-Taste kehrt man zurück in den Kommandomodus.

Von dort lassen sich mit Doppelpunkt (:) in einer Kommandozeile am Fuß des Fensters `ed`-Kommandos eingeben, z. B. `w` *Name2*, wenn man in eine andere Datei schreiben will, `w!` *Name*, wenn man die bestehende Datei überschreiben will, `q` zum Beenden nach vorherigem Schreiben, `q!` zum Beenden ohne Zurückschreiben. Üblicherweise beendet man aber den Editor `vi` mit gleichzeitigem Zurückschreiben einfacher durch Eingabe von `ZZ` (Großbuchstaben) im Kommandomodus.

Weitere Kommandos zum Suchen und Ersetzen finden sich in den Manuals. Wem auch `vi`, zu Recht, zu archaisch ist, sollte nach weiteren Editoren Ausschau halten, z. B. `joe`, `xedit`, `axe`, `xcoral`, `asedit`, oder dem bereits erwähnten `emacs`, dem wir unten einen eigenen Kommentar widmen. Auch größere Softwareentwicklungstools, wie z. B. *SoftBench* unter AIX, kommen mit modernen Editoren, die wenigstens Textmarkieren mit der Maus und *cut-and-paste* erlauben.

Emacs: Stein der Weisen oder Stein des Anstoßes?

GNU Emacs ist ein Editor, ein Textverarbeitungsprogramm, eine Softwareentwicklungsumgebung und noch viel mehr. GNU steht für „GNU is not UNIX“, ein Projekt der FSF (*Free Software Foundation*, vgl. Abschnitt 1.1) das zum Ziel hat, ein zu UNIX kompatibles, aber frei verfügbares Betriebssystem zu entwickeln. Emacs steht für *editing macros* und wurde von Richard Stallman geschrieben, dem späteren Gründer der FSF. Die Software der FSF unterliegt der GPL (*General Public License*) und ist damit „frei“, wobei sich „frei“ anders definiert als „public domain“ oder Shareware: Die Software unterliegt einem Copyright, darf aber beliebig kopiert und weitergegeben werden — auch im Quelltext —

unter der Auflage, daß neben der Wahrung des Urheberrechts auch neuerzeugte Software, sofern sie Teile der freien Software enthält, wieder frei unter den Regeln der GPL ist.

Neben diesem „moralischen“ Anspruch führt Emacs auch eine eigene Begriffswelt (Point, Mark, Region), dazu spezielle Tastenkombinationen und Fensteraufteilungen, ein. Beherrscht man sie, bzw. hat man sich an sie gewöhnt, kann man sehr schnell und flexibel verschiedenste Aufgaben bearbeiten, da Emacs zahlreiche Modi kennt, mit denen er Anwendungen unterstützt: die Bearbeitung normaler Texte, von Verzeichnissen (!), der Quelltexte vieler Programmiersprachen (Pascal, C, C++, Fortran, Tcl/Tk[1], ...), TeX[2], LaTeX[3] und HTML[4]. Weiterhin werden viele Modi für die Interaktion mit anderen Komponenten eines UNIX-Systems bereitgestellt, z. B., E-Mail[5], WWW[6], IRC[7], News[8], FTP[9], sccs und rcs[10].

An der Mächtigkeit und schieren Größe von Emacs scheiden sich denn auch die Geister. Die Gegner sagen, Emacs ist viel zu kompliziert und widerspricht der UNIX-Philosophie der kleinen, kombinierbaren Werkzeuge. Die Freunde von `emacs` argumentieren, daß die Erweiterung der Grundfunktionalität in den vielen Spezialmodi eine konsistente Bedienbarkeit aller Werkzeuge sicherstellt und daß die Einzelkomponenten von Emacs gerade der UNIX-Philosophie entsprechen.

Wie dem auch sei, die hohe Akzeptanz von Emacs bei einem großen Teil der UNIX-Gemeinde hat wiederum dazu geführt, daß einfache Emacs-Editierkommandos ihren Weg in andere UNIX-Komponenten gefunden haben. So kann man in einigen Shells (z. B. `bash`) die (Eingabe- oder) Kommandozeile mit der Tastenbelegung des Emacs editieren.

Der ursprünglich seitenorientierte Editor ist inzwischen auch mit einer graphischen Benutzeroberfläche für X11 ausgestattet, sowie für viele andere Betriebsysteme (OS/2, Windows) verfügbar. Wem der Einstieg schwerfällt, kann auf eines der zahlreichen Einführungswerke zurückgreifen [3, 5]. Hinweise zu den hier genannten und zu weiteren zahlreichen Softwaretools finden sich z. B. in dem LINUX-Buch von Strobel und Uhl [16].

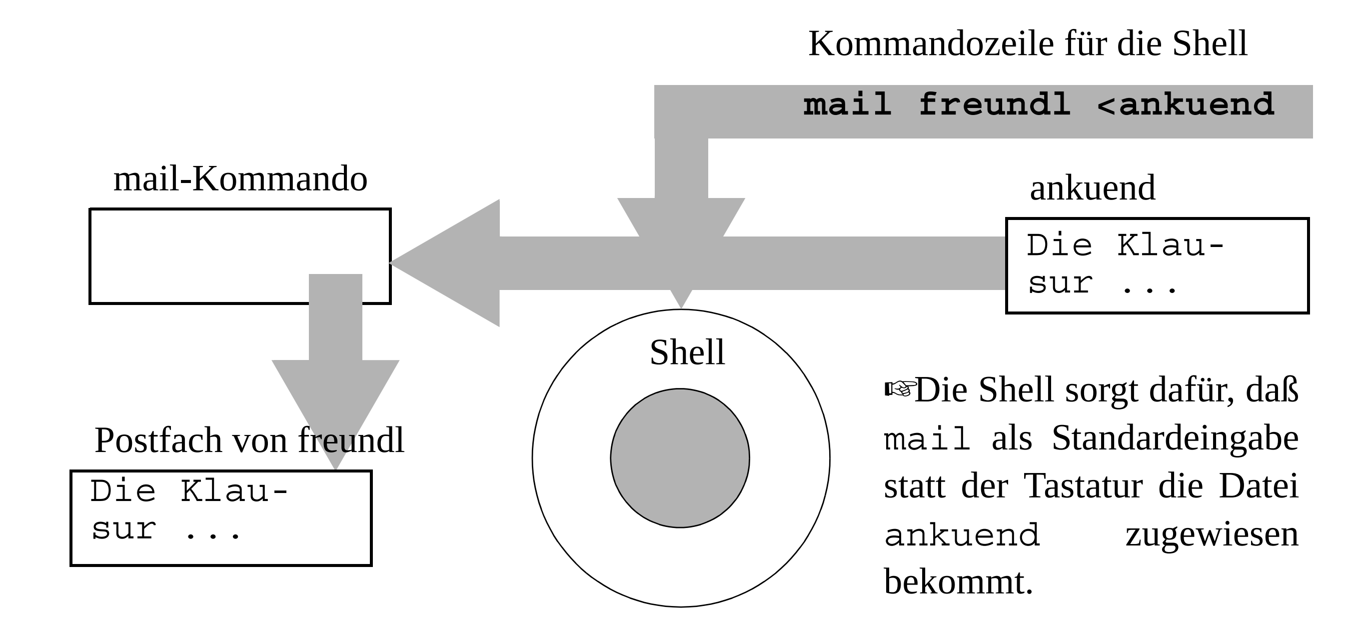
---

1. Tcl/Tk (sprich Tickel/TeKa), John Ousterhout’s Tool Command Language/Toolkit zur Erzeugung graphischer Oberflächen
2. TeX (sprich Tech), Donald E. Knuth’s Programm für die professionelle Satzgestaltung, spez. mathematischer Texte
3. LaTeX (sprich Latech) ein von Leslie Lamport erstelltes Macropaket, das den Umgang mit TeX wesentlich erleichtert; der Name ist ein mehrfaches Wortspiel: (1) der gleichnamige Kunststoff (2) TeX für den Nichtfachmann (*engl.* layman) (3) Lamport’s TeX (4) ...
4. HTML, Hyper-Text Markup Language, Formatierungssprache für Dokumente in Netzen, speziell im WWW
5. Mail, elektronische Post, siehe diesen Abschnitt oben
6. WWW, World Wide Web
7. IRC, Internet Relay Chat, System zum Plaudern im Internet, ähnlich wie News
8. News, System von Newsservern, auf denen elektronische „schwarze Bretter“ zu verschiedenen Themen eingerichtet sind und die dem Austausch von Informationen und Meinungen dienen
9. FTP, File Transfer Protocol, ursprünglich ein Dateiübertragungsprotokoll, daneben ein Programm (`ftp`), jetzt allgemein ein Oberbegriff für das Bereitstellen und Übertragen von Programmen, Dokumenten, Listen, usw.
10. `sccs` und `rcs`, source code control system und revision control system, UNIX Werkzeuge zur Pflege von Quellcode, spez. zur Versionskontrolle in Entwicklungsteams

Doch Professor Fix will nicht weiter editieren, er will jetzt die Datei `ankuend` an die Sekretärin schicken.

```
$mail freundl <ankuend
$
```

Das mail Kommando erwartet seine Eingabe von der Standardeingabe, üblicherweise der Tastatur. Durch `<ankuend` wird die Eingabe umgelenkt; sie kommt jetzt aus der Datei `ankuend`. Dies ist nur möglich, weil die Eingabe genauso wie eine Datei behandelt wird, nämlich als unstrukturierte Folge von Zeichen.



Zusammenfassung

❖ Wir haben die Post (mail) und das Postfach verwaltet (`d`,`s`,`p`,`q`,`m`), mit `ed` eine Antwort erstellt (`ed datei`, `a`,`.`,`w`,`q`) und mit `mail` Post verschickt und dabei die Standardeingabe mit dem Kleinerzeichen (<) umgelenkt.

Frage 4

```
You have mail.
$mail
From jr Fri Dec 13 13:27 GMT 1996
UMLAUF
Ich nehme an der Weihnachtsfeier teil.
Name                 Ja      Nein
-------------------------------------
S.Freundlich         x
J.Richtig            x
& s umlauf
$mail neu <umlauf
```

Wie könnte Professor Fix statt mit `s umlauf` und `mail neu <umlauf` die Nachricht unverändert an Kollegen Neu weiterleiten?

Frage 5

```
$ed zaehlen
?zaehlen
a
Werden NL-Zeichen
und
EOT-(Dateiende)Zeichen gespeichert?
.
w
59
q
```

Die richtige Antwort auf die Frage in der Datei `zaehlen` lautet:

❑ Nur NL (new-line, neue Zeile),

❑ Nur EOT (end-of-transmission, Dateiende),

❑ EOT und NL.

Frage 6

Welche der folgenden Kommandozeilen sendet die Datei `vermerk` als Post an den Teilnehmer `dekan`?

❑ `mail vermerk >dekan`

❑ `mail dekan <vermerk`

❑ `mail dekan vermerk`

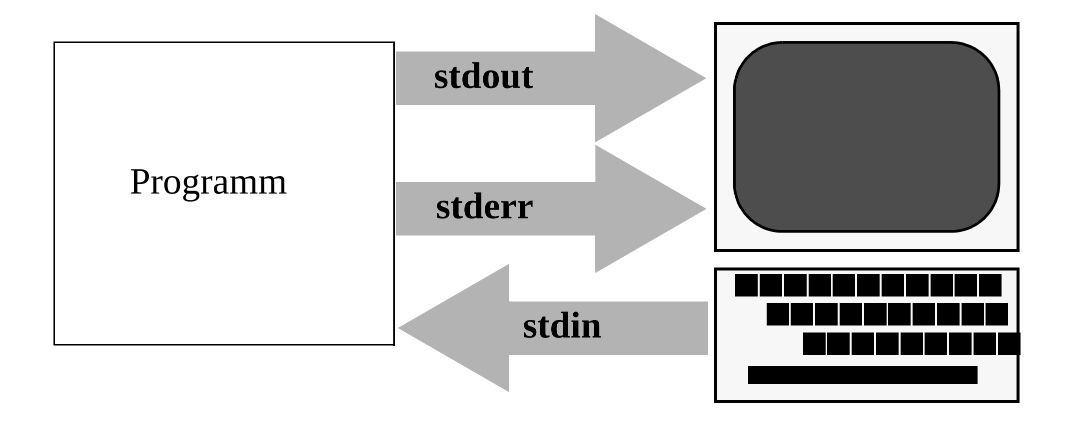
❑ `m dekan`

2.3 Umgelenkt

❖ In diesem Abschnitt wollen wir neben der Eingabeumlenkung auch die Ausgabeumlenkung und das Kommando `cat` kennenlernen.

Jedes unter UNIX ablaufende Programm ist mit drei Standarddateien verbunden:

- der *Standardeingabe* (stdin),
- der *Standardausgabe* (stdout),
- der *Standardfehlerausgabe* (stderr).



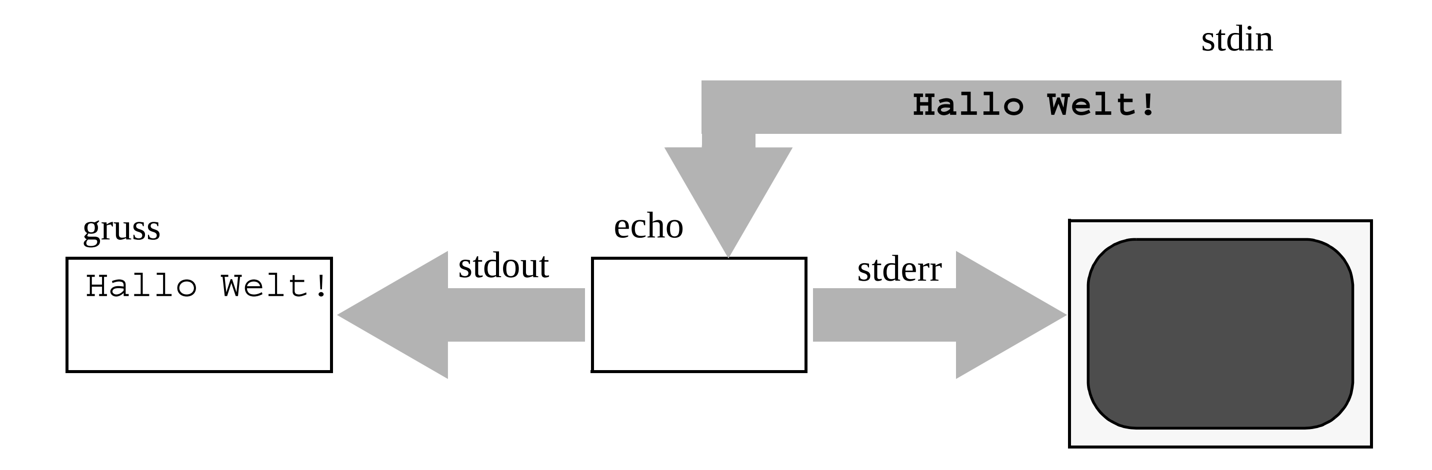
In den meisten Fällen sind stdout und stderr mit dem Bildschirm, stdin mit der Tastatur verbunden.

Im vorherigen Abschnitt haben wir die Umlenkung für die Standardeingabe kennengelernt, z. B. in `mail freundl <ankuend`. Wie die Eingabe können wir auch die Ausgabe umlenken.

```
$echo Hallo Welt! >gruss
$
```

```
$Echo Hallo Welt! >gruss
Echo: not found
$
```

Man beachte, daß die Fehlerausgabe (`Echo: not found`) weiterhin auf dem Bildschirm erscheint. Damit wird vermieden, daß Fehlermeldungen in der Ausgabedatei, hier in `gruss`, verschwinden.



Für die Ausgabedatei gilt immer:

- Ist sie bereits vorhanden, wird ihr Inhalt gelöscht.
- Ist sie noch nicht vorhanden, wird sie neu angelegt.

Mit dem doppelten Größerzeichen (>>) werden Daten an eine Datei angefügt.

Übrigens - Kommandozeilen haben ein recht freies Format. Die Shell erkennt die Zeichen `; & ( ) | < > >>` *Zeilenende Leerzeichen* und *Tabulator* als Wortbegrenzer. Statt `echo Hallo >gruss` dürfen Sie auch `>gruss echo Hallo` eingeben, denn die Shell erkennt das erste nicht mit Umlenkzeichen versehene Wort als Kommando.

Mit `cat` (con**cat**enate and print) können wir eine Datei ausgeben.

```
$echo Hallo Welt >gruss
$cat gruss
Hallo Welt!
$echo Umlenken, nicht Ablenken >>gruss
$cat gruss
Hallo Welt!
Umlenken, nicht Ablenken
```

Alles klar? Dann noch dieses kleine Rätsel, das allerdings bei neueren UNIX-Systemen nicht von der Shell akzeptiert wird:

```
$cat gruss >gruss
$cat gruss
$
```

Bei `cat gruss >gruss` entsteht eine leere Datei, denn das Ziel, hier gleichzeitig die Quelldatei, wird zuerst von der Shell gelöscht, dann kopiert. Liefert `cat gruss >>gruss` eine Endlosschleife? Nein, denn die Quelldatei wird vor dem Kopieren festgelegt, bzw. die Shell läßt generell keine gleichen Ein- und Ausgabedateien zu, z. B. in AIX.

Das Kommando `cat` liest sequentiell die als Argumente genannten Dateien und schreibt sie auf die Standardausgabe. Der Name `cat` (concatenate) rührt von der Anwendung

```
cat datei1 datei2 >datei3
```

her, d. h. `datei1` wird mit `datei2` verkettet und in `datei3` abgelegt, wie z. B. in

```
$echo P.S. Dauer drei Stunden. >ps
$cat ankuend ps >neuankuend
$
```

Die folgende Anwendung zeigt ein weiteres Beispiel zum Anfügen an Dateien.

```
$cat umlauf
From jr Fri Dec 13 13:27 GMT 1996
UMLAUF
Ich nehme an der Weihnachtsfeier teil.
Name                Ja      Nein
-----------------------------------
S.Freundlich        x
J.Richtig           x
$echo F. Fix.            x >>umlauf
$
```

Professor Fix hat seine Zeile an den Umlauf angehängt.

Zusammenfassung

❖ Wir haben die drei Standarddateien *stdin*, *stdout*, *stderr* und ihre Umlenkung durch < , > und >> kennengelernt. Mit `cat` *datei* ... können wir Dateien verkettet in die Standardausgabe schreiben.

Frage 7

Wie lautet die Ausgabe am Bildschirm, wenn die Shell `sh` die unten aufgeführte Datei `grussaus` zur Interpretation und Ausführung erhält?

```
$echo Hallo Leute! >gruss
$echo cat gruss >grussaus
$cat grussaus
cat gruss
$sh grussaus
????????????
$
```

Frage 8

Wie kann Professor Fix an die Datei `raum` mit Inhalt

```
From freundl Mon Dec 16 14:20 GMT 1996
Raum 1409 fuer UNIX-Klausur 14/02/97,
14-16 Uhr bestätigt.
```

die Zeilen „gesehen“ und das aktuelle Datum anfügen?

❏ cat raum gesehen date >>raum

❏ echo gesehen date >>raum

❏ echo gesehen >>raum
date >>raum

❏ echo gesehen >>raum
cat date >>raum

Frage 9

Auch wenn freies Format für Kommandozeilen zugestanden wird, welche Zeile kann nicht gutgehen?

❏ >aus <ein cat

❏ <ein >> aus cat

❏ >gruss Hallo Leute echo

LEKTION 3:
Kommandos im Zusammenspiel

3.1 Optionen

❖ In diesem Abschnitt wollen wir einige Kommandos zur Druckausgabe (`lpr`), zum Zählen (`wc`), zeichenweisen Ersetzen (`tr`) und Sortieren (`sort`) zusammen mit dem Gebrauch von Optionen zeigen. Daneben werden die Zusammenfassung mehrerer Kommandos in einer Zeile und das Starten im Hintergrund vorgestellt.

In der vorherigen Lektion hat Professor Fix eine Datei `ankuend` angelegt. Diese Datei möchte er sich noch einmal anschauen und dann ausdrucken.

```
$cat ankuend
Die UNIX-Klausur findet am Freitag, dem 14. Februar 1997,
um 14 Uhr im Raum 1409 statt.
Hilfsmittel sind nicht zugelassen.
        gez. Fix 13/12/1996
$lpr -rm ankuend
$
```

Mit `lpr -rm ankuend` wird die Datei ankuend in die *Warteschlange des Druckspoolers* eingetragen und bei Verfügbarkeit des Druckers ausgegeben. Diese Sequentialisierung der Druckdateien ist für den Mehrprogrammbetrieb notwendig.

Die Optionen des Kommandos `lpr` sind:

- `r` (**r**emove) Löschen der Datei nach Eintrag der Kopie in die Warteschlange.
- `m` (**m**ail) Eine Mitteilung nach Beendigung des Druckens soll erfolgen.
- `n` (**n**o mail) Keine Mitteilung.
- `c` (**c**opy) Lege Kopie in Warteschlange ab. Original sofort verfügbar.

Optionenangaben sind meist Einzelbuchstaben mit einem vorangestellten Minuszeichen.

Die Ankündigung der Klausur ist nun also heraus, die Datei `ankuend` ist gelöscht (Option `r`) und nach der Beendigung des Druckvorgangs wird eine Mitteilung an Professor Fix geschickt (Option `m`).

So, jetzt zur Post!

```
$mail
From gabi Tue Dec 17 18:40 GMT 1996
Wir haetten gerne noch je ein Skript nachbestellt.
UWE
BERND
GABI
&
```

Fix rettet den Brief in die Datei `mehrbest` und beendet `mail`.

```
& s mehrbest
& q
$
```

Die letzten drei Zeilen möchte er der Datei `nachb` anfügen, die er sich zuerst einmal anschaut.

```
$cat nachb
klaus
monika
werner
bernd
$tail -3l mehrbest >>nachb
$
```

Zum Anfügen verwendet Professor Fix das Kommando `tail` (Ende einer Datei ausgeben).

Die Syntax ist `tail` [`+-`*zahl* [`bcl`]] [*datei*], wobei

- `+` Abstand vom Anfang der Datei,
- `-` Abstand vom Ende der Datei,

- `b` (blocks) in Blöcken,
- `c` (characters) in Zeichen,
- `l` (lines) in Zeilen

bedeutet.

Die mit `tail` verlängerte Datei `nachb` schaut sich Professor Fix wieder an.

```
$cat nachb
klaus
monika
werner
bernd
UWE
BERND
GABI
$
```

Wieviele Bestellungen er nun hat, kann er durch Zählen der Zeilen der Datei `nachb` feststellen. Dazu dient das Kommando `wc` (**w**ord **c**ount).

```
$wc nachb
 7 7 41 nachb
$
```

Das Kommando `wc` [`-lwc`] [*datei* ...] zählt (mit `-l` nur) die Zeilen, (mit `-w` nur) die Wörter und (mit `-c` nur) die Zeichen der Datei *datei*. `nachb` enthält also 7 Zeilen, 7 Wörter und 41 Zeichen.

Unter Wörtern versteht man in UNIX immer maximale Folgen von Zeichen, begrenzt durch ein *Leerzeichen*, *Tabulator* oder *Zeilenende* (newline), die kollektiv als *white spaces* bezeichnet werden.

Einige der Namen in der Liste der Nachbesteller sind mit Großbuchstaben geschrieben. Dieses ändert Professor Fix mit dem Kommando `tr` (**tr**anslate).

```
$tr [A-Z] [a-z] <nachb >nachb2
$
```

Mit `tr` [`-cds`] *kette*1 *kette*2 werden Zeichen der Standardeingabe in Zeichen der Standardausgabe unter Substitution von Zeichen aus *kette*1 in Zeichen aus *kette*2 kopiert. Dabei bedeutet

- `c` (**c**omplement) nimmt das Komplement zu den Zeichen in Zeichenkette 1.
- `d` (**d**elete) entfernt in Zeichenkette1 vorkommende Zeichen bei Übertragung.
- `s` (**s**queeze) komprimiert alle Folgen von gleichen Zeichen in Zeichenkette 2 zu einem Zeichen bei der Ausgabe.

Jetzt sollten also Groß- durch Kleinbuchstaben substituiert sein, wobei das Resultat in `nachb2` stehen müßte. Professor Fix schaut sich `nachb2` an und zählt nochmals die Zeilen dieser Datei.

```
$cat nachb2; wc nachb2
klaus
monika
werner
bernd
uwe
bernd
gabi
 7 7 41 nachb2
$
```

Natürlich hat sich nichts an der Größe der Datei geändert. Er vermißt aber das Bereitzeichen `$` zwischen der Liste und der Zählung. Was ist los?

Die Kommandos `cat nachb2` und `wc nachb2` erscheinen in einer Zeile, getrennt durch ein Semikolon, wie oben zu sehen ist. Die Shell bricht die Zeile in einzelne Kommandos auf und arbeitet sie sequentiell von links nach rechts ab. Erst nach Beendigung des letzten Kommandos erscheint dann wieder das Bereitzeichen `$`.

Möchte man Kommandos (Programme) gleichzeitig arbeiten lassen, kann man durch Anhängen des Zeichens `&` (ampersand, Und-Zeichen) ein Kommando als *Hintergrundprozeß* starten. Das Bereitzeichen `$` erscheint dann sofort, und neue Kommandos können ausgeführt werden. Die Hintergrundverarbeitung ist daher günstig für länger laufende Programme *ohne Interaktion*.

Dazu ein Beispiel: Professor Fix sortiert seine Bestellungen und läßt sich das Datum zeigen.

```
$sort nachb2 >>nachb & date
3121
Wed Dec 18 11:05:55 GMT 1996
$
```

Das Kommando `sort` läuft hier als Hintergrundprozeß, was natürlich bei so wenigen Bestellungen nicht nötig gewesen wäre. Wie das Semikolon ist auch das Zeichen `&` ein *Kommandotrenner*. Die gezeigte Zahl 3121 ist die Nummer des Prozesses, der `sort` ausführt. `kill 3121` würde den Prozeß `sort` stoppen.

```
$from lpr ankuend printed
$
```

Jetzt ist auch die noch ausstehende Mitteilung angekommen, daß die Datei `ankuend` gedruckt ist. Das hat ja auch lange genug gedauert. Da hat wohl wieder jemand ganze Bücher ausdrucken lassen.

Mehr oder weniger komfortables Blättern

Sich mit `cat` eine Datei am Bildschirm anzusehen geht eigentlich nur, wenn die Datei sehr klein ist. Bei größeren Ausgaben „rauscht“ der Text durch und das Anhalten mit ‘CTRL-s’ und Weitermachen mit ‘CTRL-q’ ist auch mehr etwas für schnelle Finger.

Für komfortableres Blättern empfiehlt sich die Ausgabe mit `more` *datei*, wobei `more` nach jeder Seite anhält und mit der Leertaste dazu gebracht wird, eine Seite weiterzublättern. Noch komfortabler ist das von GNU stammende Kommando `less`.

Neben diesen Kommandos, die das Auslesen der Zeilen einer Datei kontrollieren, kann man sich in einem komfortablen Fenstersystem darauf abstützen, daß die Bildschirmhistorie nach einem `cat` lange genug zurückreicht, so daß man mit einem Rollbalken am Fenster vor und zurückblättern kann.

Zusammenfassung

❖ Wir haben die Kommandos `lpr`, `wc`, `tr` und `sort` und einige (wenige) ihrer Optionen kennengelernt. Mit dem Semikolon können wir die sequentielle Abarbeitung mehrerer Kommandos erreichen. Mit dem Zeichen `&` können wir Kommandos als Hintergrundprozesse laufen lassen.

Frage 1

Mit `tr -? '\012' < textfile > newfile` sollen mehrere aufeinanderfolgende Neue-Zeile-Zeichen (oktal 12) in einen einzigen Zeilenvorschub zusammengeschoben werden. Was muß statt `-?` als Option stehen?

Frage 2

Welche der vier Schreibweisen ist (sind) gültig, wenn die Optionen `r` und `n` gesetzt werden sollen?

❏ `lpr -rn ankuend`

❏ `lpr -r n ankuend`

❏ `lpr -r -n ankuend`

❏ `lpr -nr ankuend`

3.2 In die Röhre geschaut

❖ In diesem Abschnitt wollen wir die Pipelineverarbeitung und das Filterkonzept behandeln. Wir verwenden die Kommandos `tee` und `uniq`.

Professor Fix findet, daß der Rechner heute besonders langsam ist. Sind so viele Teilnehmer angemeldet? Er sieht nach.

```
$who >teiln
$wc -l teiln
 6 teiln
$rm teiln
```

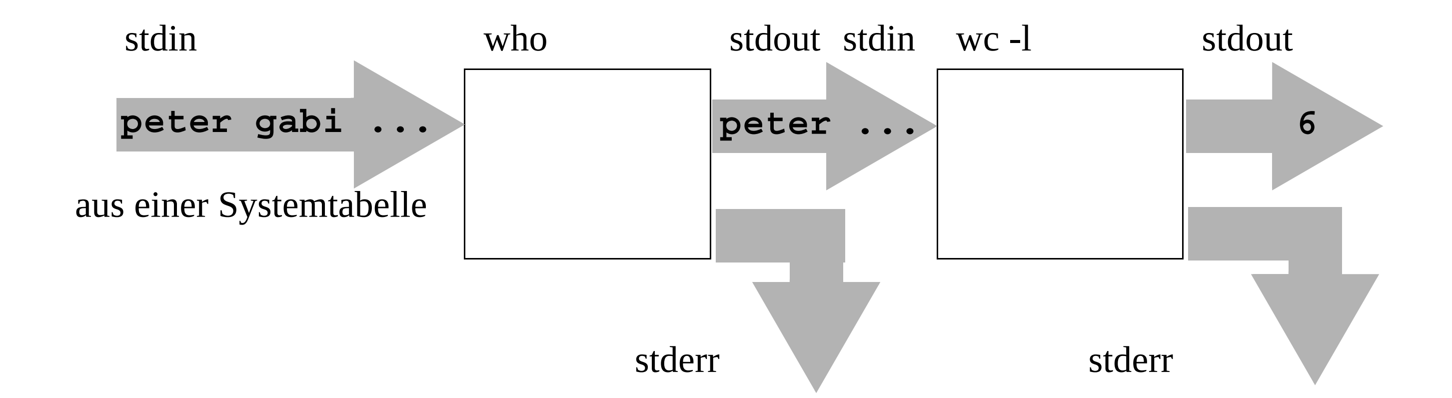
Nein, nur 6 Teilnehmer. Die Folge der Kommandos

- Ausgabe in Hilfsdatei `teiln`
- Zeilen in `teiln` zählen
- Datei `teiln` mit `rm` (**rem**ove) entfernen

ist recht aufwendig, da eine unnötige Hilfsdatei erzeugt wird. Einfacher und in einer Zeile geht es mit einer *Pipe (engl*. pipe: Rohr, Röhre).

```
$who | wc -l
 6
$
```

Das *Pipesymbol* (der senkrechte Strich) zwischen zwei Kommandos, z. B. in `who | wc`, verbindet die *Standardausgabe* des ersten Kommandos (`who`) mit der *Standardeingabe* des zweiten Kommandos (`wc`). Die einzelnen Kommandos einer Pipeline werden als nebenläufige Prozesse gleichzeitig abgearbeitet.



Professor Fix möchte jetzt die sortierte Liste der Bestellungen seines Skriptes kontrollieren und drucken.

```
$cat nachb
bernd
```

```
bernd
gabi
klaus
monika
uwe
werner
$
```

Ein Name ist doppelt. Gleiche aufeinanderfolgende Zeilen lassen sich mit dem Kommando `uniq` herausfiltern. Wir setzen deshalb den Filter `uniq` vor die Druckausgabe.

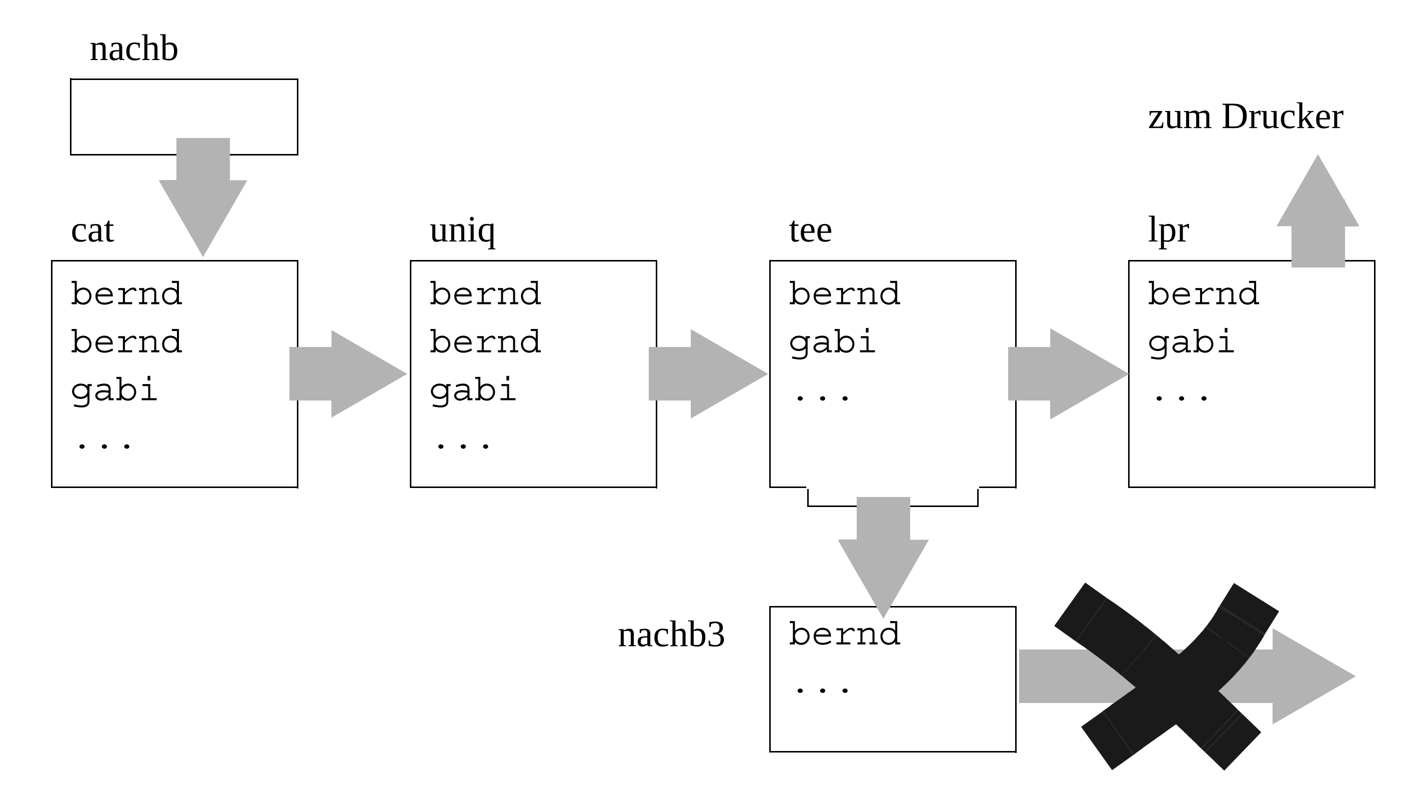
```
$cat nachb | uniq | lpr
$
```

Statt `cat nachb | uniq | lpr` hätten wir auch `uniq <nachb | lpr` schreiben können.

Eigentlich wollte Professor Fix auch eine Kopie der Liste als Datei halten. Dazu hätte er ein *T-Stück* in die Pipeline einbauen können.

```
$cat nachb | uniq | tee nachb3 | lpr
$
```

Das Kommando `tee` [*datei*] macht eine Kopie der aus der Standardeingabe in die Standardausgabe geleiteten Zeichen. Die mit `tee` kopierten Daten können aber nicht direkt mit einer weiteren Pipe verarbeitet werden.



Pipelinebildung und *Filterkonzept* sind neue Paradigmen der Informationsverarbeitung. Sie können in vielen Anwendungen eingesetzt werden, z. B. in mehrfach-aktiven verbundenen Fenstern, die dann als *Erzeuger-Verbrauchersysteme* Daten produzieren (wie `cat` oben) und konsumieren (wie `lpr` oben).

Alles im Fluß

Durch eine Pipe wird nicht nur die Ausgabe des Kommandos links vom Pipesymbol mit der Eingabe des Kommandos rechts vom Pipesymbol verbunden, auch der Ablauf der Kommandos (genauer: der dahintersteckenden Prozesse, siehe nächste Abschnitte) wird auf sehr einfache Art geregelt: das zweite Kommando kann erst etwas lesen, wenn das erste Daten in die Pipe geschrieben hat.

Weil dieses Konzept so einfach und mächtig ist, könnte man versucht sein, es zur Grundlage beliebiger Kommunikation zwischen Programmen (und Benutzern?) zu machen. Schon *Ritchie* und *Thompson* hatten 1978 angemerkt [11], daß die Linearität von Pipes nicht „gottgewollt" ist sondern eher den Zwängen der linearen Syntax der Shell genügt.

Dem steht zunächst der Erzeugungsmechanismus für Pipes entgegen: Er verlangt, daß die beteiligten Kommandos einen gemeinsamen erzeugenden Vater haben, in den Beispielen oben den Shell-Prozeß, der die Standardeingabe des einen mit der -ausgabe des anderen verbindet. Für eine allgemeine „Prozeßkommunikation" kann man dieses verwandtschaftliche Verhältnis aber nicht voraussetzen.

Behandelt man dagegen die Pipe wie eine temporäre Datei, kann man einen uni-direktionalen Datenstrom zwischen nahezu beliebigen Kommandos aufbauen. Dies ist die Grundidee der benannten Pipe (*named pipe*, *FIFO*). Mit dem Kommando `mknod` *name type* läßt sich eine Spezialdatei mit Namen *name* anlegen, in unserem Fall für `test1` durch Angabe des Typs `p` eine benannte Pipe.

In einem Mehrfenstersystem kann man sich jetzt Daten von einem Fenster in ein anderes schicken, im Beispiel die Ausgabe von `who`.

```
$mknod test1 p
who >test1
$     Promptzeichen erst nachdem das
      zweite Fenster fertiggelesen hat!
```

```
$cat <test1
fix hft/0 Sep 05 09:46
fix pts/0 Sep 18 15:23
fix pts/1 Sep 24 09:06
$rm test1
$
```

Ohne mehrere Fenster muß man die Umlenkung als Hintergrundprogramm starten (siehe folgende Abschnitte), da der Absender blockiert wird und auf die Aktivierung des Empfängers wartet.

Named pipes oder FIFOs (first in first out Pipes), wie sie z. B. im POSIX-Standard heißen, werden im Dateisystem abgelegt. Daraus ergeben sich schon einige Einschränkungen, etwa für die Nutzung über Rechnergrenzen hinweg. Daneben ist ein korrektes Verhalten in einer Client-Server-Umgebung, also mit mehreren lesenden oder schreibenden Prozessen (Kunden) und einem bedienenden Prozeß (dem Auftragnehmer) schwierig herzustellen, da named pipes prinzipiell wie die normale Pipe uni-direktional sind und keine eigene Mechanismen für die Prozeßsynchronisierung (siehe nebenläufige Prozesse im Abschnitt 3.3 unten) bereitstellen.

Fortschrittlichere Kommunikationsmöglichkeiten zwischen Prozessen (IPCs, Inter Process Communications) werden in Lektion 10 kurz angesprochen. Details finden sich z. B. in dem Werk von *Stevens* [15].

Zusammenfassung

❖ Wir haben das Pipeline- und Filterkonzept unter Verwendung des Pipeoperators kennengelernt. Zur Anwendung kamen die Kommandos `rm` zum Entfernen einer Datei, `uniq` zur Duplikatsbehandlung und `tee` zur Kopienbildung in Pipelines.

Frage 3

Was erzeugt das Kommando `who >lpr`?

❏ Eine gedruckte Liste der momentanen Teilnehmer.

❏ Eine Datei `lpr` mit den momentanen Teilnehmern.

❏ Die Ausgabe der am Drucker wartenden Jobs.

❏ Eine Fehlermeldung.

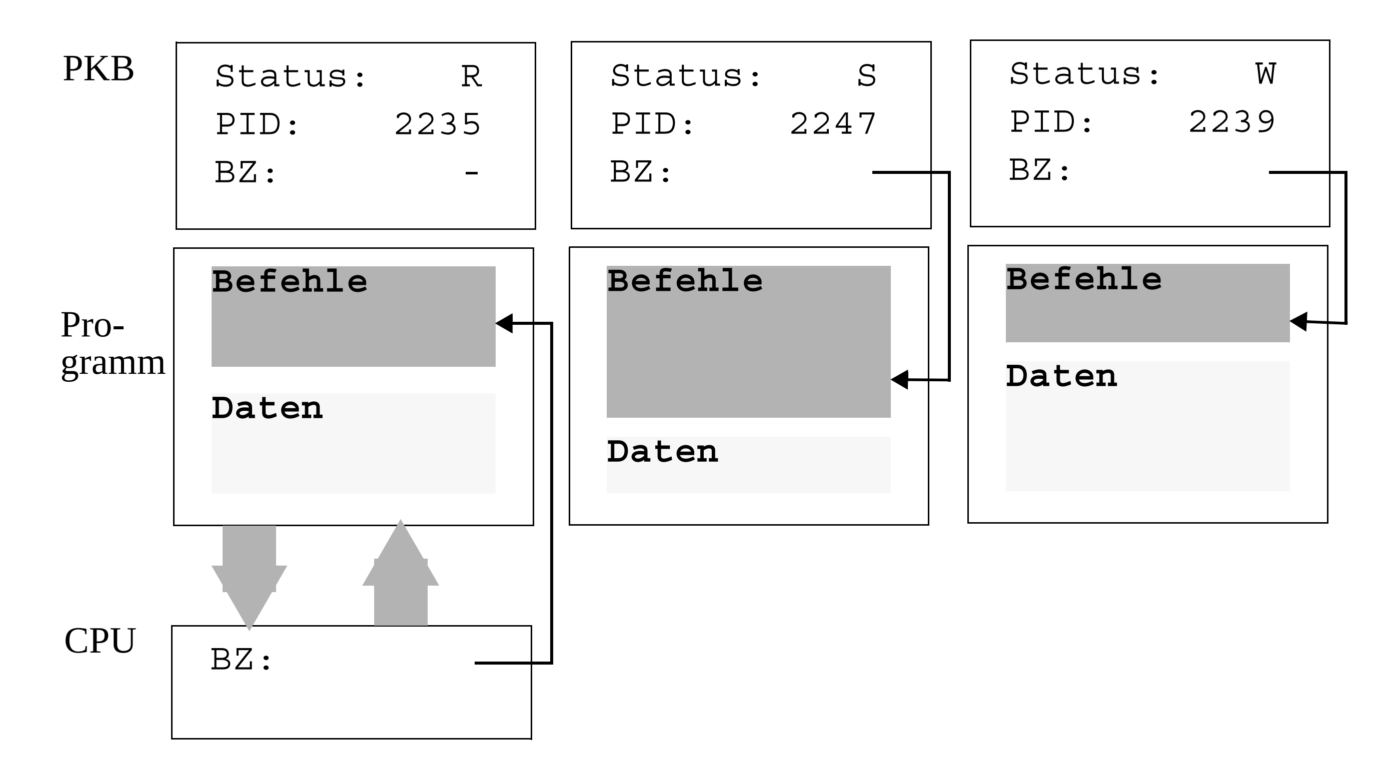
Frage 4

Was liefert die Pipeline `who | wc | wc -l` ?

3.3 Kurzer Prozeß

❖ In diesem Abschnitt erläutern wir kurz den Unterschied zwischen Programmen und Prozessen und die Übergänge zwischen rechnend, bereit und wartend.

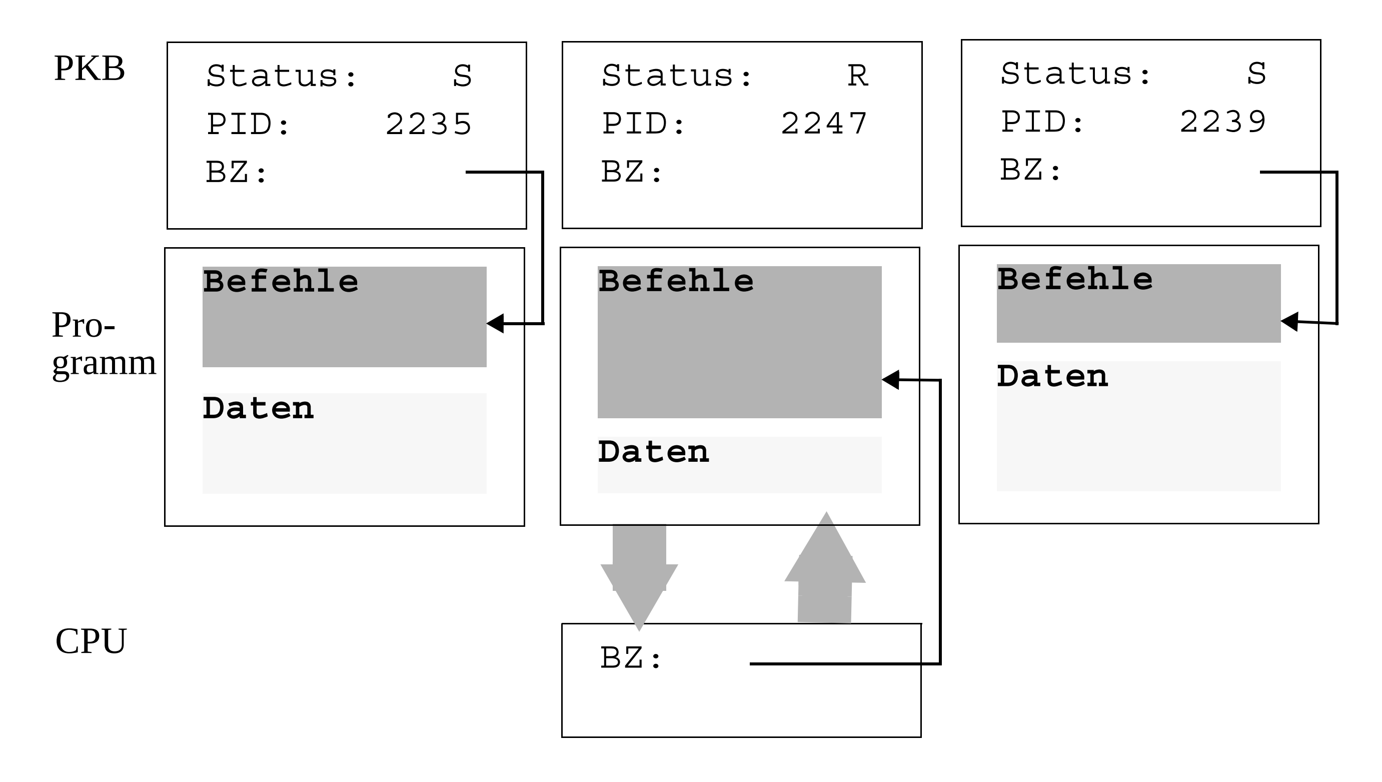
Nicht nur Juristen sprechen von *Prozessen*, auch Informatiker. Sie meinen damit aber Programme, die sich in Bearbeitung befinden.



Damit ein Prozeß aktiv werden kann, braucht er u .a. den Prozessor (die CPU), um die Befehle des Programms nacheinander zu verarbeiten. Ist ein Betriebssystem *mehrprogrammfähig*, wie z. B. UNIX, wechselt der Prozessor zwischen den Prozessen so schnell hin und her, daß der Eindruck entsteht, sie würden parallel ablaufen.

UNIX gehört zu den Betriebssystemen, die ihr Prozeßkonzept nicht verstecken. Wir können deshalb hinter die Kulissen schauen.

Im ersten Bild ist ein Programm in Abarbeitung. Der *Befehlszähler* BZ gibt den gerade verarbeiteten Befehl an. Der *Prozeßkontrollblock* PKB hält die Verwaltungsinformationen über den Prozeß fest, z. B. die *Prozeßnummer* PID (2235).

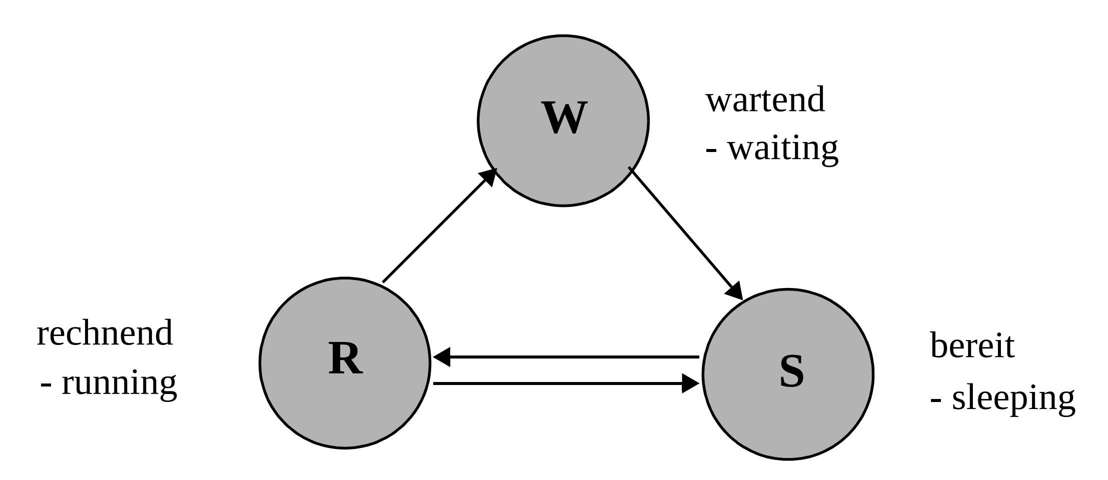


Zwei weitere Prozesse mit den Prozeßnummern 2247 und 2239 sind im Rechner, werden aber momentan nicht bearbeitet.

Im darauf folgenden Bild hat die CPU zu Prozeß 2247 gewechselt, z. B. weil jeder Prozeß reihum nur eine feste Zeit von 0,1 s zugeteilt bekommt. Zugleich hat sich der Status von 2235 zu *bereit* (S, *sleeping*) und von 2247 zu *rechnend* (R) geändert. Inzwischen hat sich auch der Status bei 2239 von *wartend* (W, *waiting*) auf bereit (S) geändert, ohne daß die CPU zugeteilt war.

Falls der Prozeß 2247 in seinem Bearbeitungsintervall einen Lesebefehl erreichen sollte, gibt er die CPU sofort ab und wartet er auf eine Tastatureingabe im Zustand W.

Dieses Spiel kann in UNIX auf leistungsfähigen Rechenanlagen viele hundertmal in der Sekunde erfolgen. Die Übergänge zwischen rechnend, bereit, und wartend werden vom Bertriebssystemkern (Kernel) gesteuert.



Nicht im Bild der Prozeßübergänge sind die vier weiteren möglichen Zustände 0 (nicht vorhanden), Z (beendet), I (Zwischenzustand) und T (angehalten).

Zusammenfassung

❖ Wir haben das Konzept der CPU-Umschaltung zwischen Prozessen kennengelernt und etwas über die Zustandsübergänge zwischen rechnend, bereit und wartend erfahren.

Frage 5

Die Umschaltung der CPU zwischen den Prozessen erledigt eine Routine im Betriebssystemkern mit dem Namen *dispatcher*. Sie sorgt u. a. dafür, daß

❑ ein wartender Prozeß die CPU bekommt,

❑ ein bereiter Prozeß die CPU bekommt,

❑ einem rechnenden Prozeß die CPU entzogen wird.

Welche der Aussagen ist (sind) **falsch**?

Frage 6

Im Gegensatz zu CP/M und MS-DOS war UNIX von Anfang an auch ein Mehrbenutzerbetriebssystem. Dies setzt auch Mehrprogrammfähigkeit voraus. Ist dies richtig oder haben die beiden Aussagen nichts miteinander zu tun?

3.4 Was so alles läuft

❖ In diesem Abschnitt lernen wir das Auftreten von Prozessen in UNIX kennen, ihre Anzeige mit `ps` und ihre Beeinflussung mit `nice`, `nohup` und `at`.

Wo kommt ein Benutzer in Kontakt mit einem Prozeß?

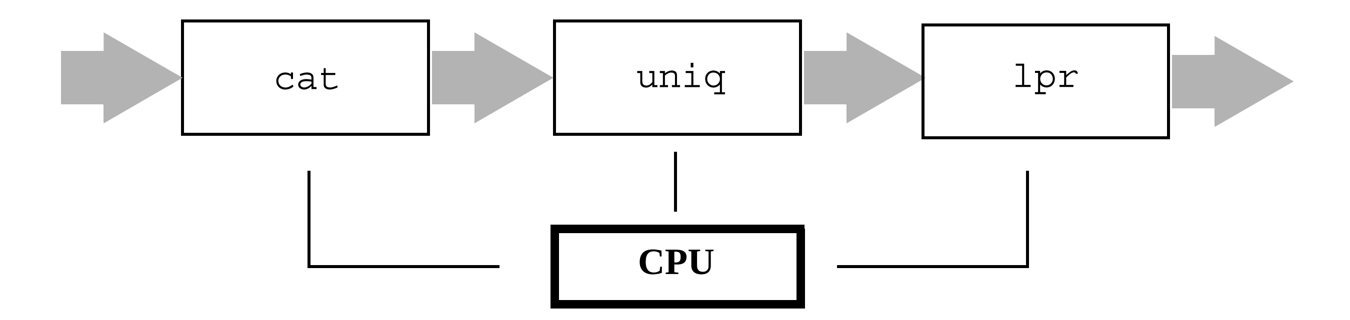
Die Antwort lautet: **überall**!

Noch bevor man angemeldet ist, sind Prozesse dabei, die Terminals zu aktivieren (`init`). Außerdem wartet ein Prozeß `getty` auf die Anmeldung und ein anderer bedient sie - das uns bekannte `login`.

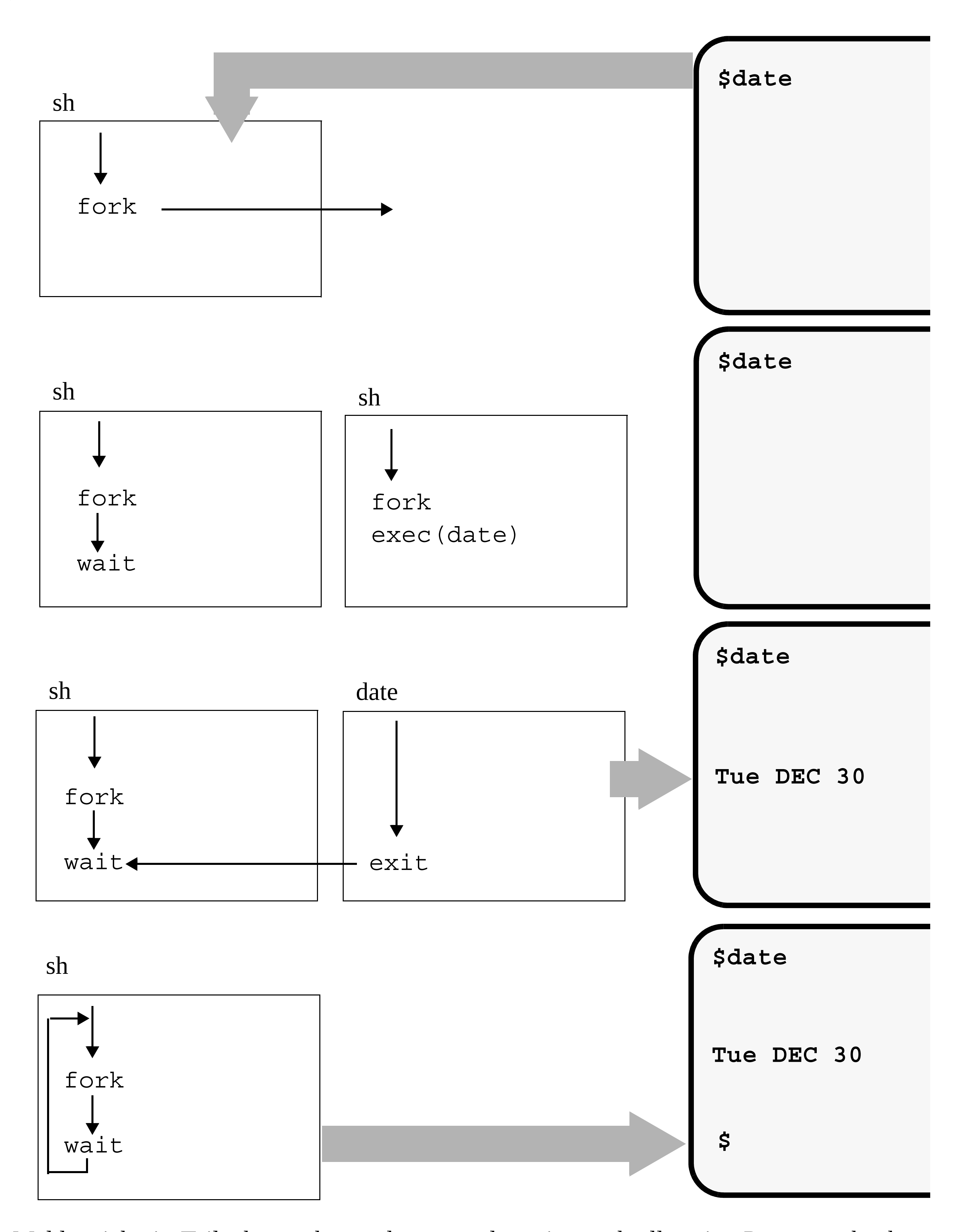
Nach geglückter Anmeldung startet sofort ein Prozeß für den Kommandointerpreter, die *Shell*. Kommt ein Kommando zur Ausführung, verzweigt (`fork`) die Shell und erzeugt damit einen weiteren Prozeß, der das Kommando verarbeitet. Der Shell-Prozeß wartet solange auf die Beendigung des Kommandos.

Die folgenden Bilder zeigen den schematischen Ablauf bei Eingabe des Kommandos `date` mit den Systemaufrufen des Betriebssystemkerns. Die technischen Details müssen wir hier übergehen.

Mehrere Prozesse werden (quasi-) gleichzeitig gestartet und abgearbeitet, wenn man zwei oder mehrere Kommandos zu einer Pipe verbindet, wie z. B. in `cat nachb2 | uniq | lpr`, das drei nebenläufige Prozesse aktiviert.

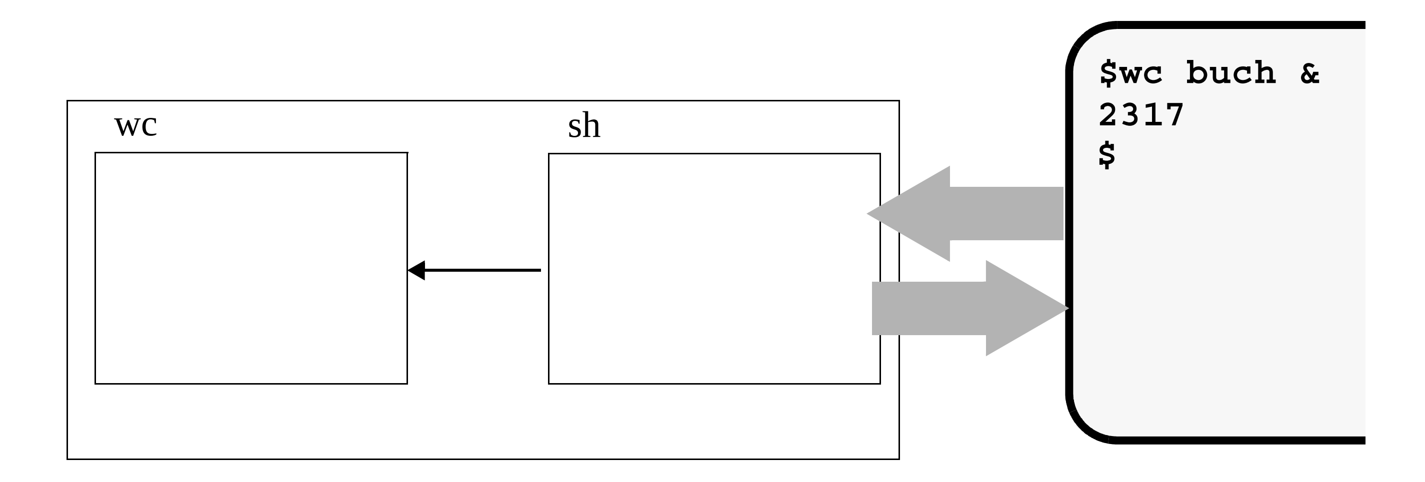


Auch beim Starten einer Hintergrundverarbeitung, wie z. B. in `wc buch &` wird ein neuer Prozeß geschaffen. Allerdings wartet die Shell nicht auf die Beendigung, sondern meldet sich sofort wieder.



Meldet sich ein Teilnehmer ab, werden normalerweise auch alle seine Prozesse abgebrochen, in obigem Fall also die Shell und `wc`.

Will man langlaufende Kommandos mit `&` als Hintergrundprozeß starten, ohne auf deren Beendigung warten zu müssen, so kann man dies mit `nohup` *kommando* `&` erreichen (**no**



hang**up**, nicht aufhängen). Eine eventuelle Ausgabe wird in der Datei `nohup.out` gerettet. Nachträglich läßt sich `nohup` leider nicht angeben.

Hat man ein Programm, das den Rechner stark belastet, sollte man den Mitbenutzern gegenüber nett sein und die Priorität durch `nice` [-*zahl*] *kommando* heruntersetzen.

```
$nice sort <index >sindex &
$
```

Alternativ kann man das Kommando auch mit `at` oder dem Kommando `batch` zu nachtschlafender Zeit oder am Wochenende ausführen lassen.

```
$at 1530 -f termin
$
```

Das Kommando `at` wird mit einem *Datum* oder einer *Uhrzeit* (1530 entspricht 15 Uhr und 30 Minuten) aufgerufen. Was ausgeführt wird, gibt man auf der Standardeingabe ein (Ende mit CTRL-d) oder als Shell-Skript mit der Option `-f`. In unserem Beispiel enthält die Datei `termin` demnach die Anweisungen, die zu dem angegebenen Zeitpunkt ausgeführt werden sollen. Professor Fix schaut sich den Inhalt von `termin` nochmal an.

```
$cat termin
echo Sitzung in 20 Minuten >/dev/tty
$
```

Die Standardausgabe und Standardfehlerausgabe der Kommandos, die in der Datei aufgeführt werden, werden bei Verwendung von `at` in `mail` weitergeleitet, wenn man sie nicht umleitet, was man bei größeren Ausgaben in der Regel vorsehen wird. Mit `>/dev/tty` hat Fix dafür gesorgt, daß die Ausgabe von `echo` auf seinem Terminal erscheint.

Auch für das Betriebssystem selbst arbeiten eine Reihe von Prozessen, z. B. der sogenannte Druckerdämon `lpd`, der durch das Benutzerkommando `lpr` geweckt wird und die Spool-Dateien ausdruckt.

Weitere, im Hintergrund periodisch ablaufende Programme sind `update`, das alle 30 Sekunden die Ausgabepuffer auf die Platte zurückschreibt, und `cron`, das jede Minute in einer Tabelle nach Einträgen sucht, die zu diesem Zeitpunkt ausgeführt werden sollen.

Auf vielen Anlagen ist auch der `swapper` installiert, ein Prozeß mit Nummer 0 für das Ein- und Auslagern von Programmen im Hauptspeicher. Welche Prozesse es gibt, kann man sich mit dem Kommando `ps` (report **p**rocess **s**tatus) anzeigen lassen.

```
$ps -e | tee gruss | lpr
$cat gruss
PID  TTY     TIME    CMD
  0    ?     0:28    swapper
267   co     0:09    -sh
241   11     1:47    /usr/games/chess
 30    ?     0:09    /etc/cron
385   co     0:01    ps -e
386   co     0:00    lpr
387   co     0:00    tee gruss
$
```

So sieht es (wie üblich vereinfacht) bei Professor Fix aus, wenn er `ps -e | tee gruss | lpr` eingibt. Er sieht alle Prozesse (Option `e`), die momentan auf dem Rechner ablaufen — auch die der anderen Anwender. (Was läuft denn da nur am Terminal Nr. 11 so lange?)

Etwas für neugierige Menschen — nicht wahr?

Auftragssteuerung

Etwas neuere Shells als die Bourne-Shell, z. B. die Korn- (`ksh`) und die Bourne-Again-Shell (`bash`) bieten mehr Möglichkeiten, interaktive Vordergrund- und Hintergrund-(Batch-)prozesse zu steuern. Ein geeignetes Kommando, unabhängig von der verwendeten Shell, zum Testen der Möglichkeiten ist `sleep` *sekunden*, das einen Prozeß erzeugt, ihn die angegebene Anzahl von Sekunden schlafen läßt und ihn dann ohne weitere Wirkung beendet. Eine mögliche Anwendung ist die Benachrichtigung aller Teilnehmer (Kommando `wall`) über das bevorstehende Herunterfahren des Rechners:

```
(
 echo "SYSTEM ABSCHALTUNG IN 10 MINUTEN!" | wall
 sleep 300; echo "SYSTEM ABSCHALTUNG IN 5 MINUTEN!" | wall
 sleep 240; echo "SYSTEM ABSCHALTUNGS IN 1 MINUTE!" | wall
 sleep 60; shutdown
 )&
```

Mit ‘CTRL-z’, dem *susp-Zeichen*, kann man ferner einen gerade laufendes Kommando anhalten (*engl.* suspend) und den zugehörigen Prozeß damit in den Zustand `T` (*stopped*) versetzen. Ob das susp-Zeichen für das Terminal gesetzt ist, erfahren Sie neben vielen anderen Informationen mit dem Befehl `stty -a`. Ob die Auftragsverwaltung generell

möglich ist, also die folgenden `fg` und `bg` Kommandos, erfährt man mit `set -o`, wobei `monitor on` angezeigt werden sollte.

Vorausgesetzt, man kann ein zweites Fenster (oder zweites Terminal eines Teilnehmers) am Rechner aufmachen, kann man jetzt mit z. B. `sleep 60` im ersten Fenster und mit `ps -le` im zweiten Fenster den Zustand der Prozesse (inklusive `sleep` mit Zustand `S`) sehen, danach mit 'CTRL-z' das Kommando `sleep` anhalten und wieder mit `ps -le` den Wechsel in den Zustand `T` beobachten. Als längerlaufendes Kommando wählen wir rekursives Auflisten (`ls -R`) des Vorgängerverzeichnisses (`..`), umgelenkt in das „leere Gerät“ `/dev/null`, scherzhaft auch der große Biteimer genannt.

Alternativ kann man sich mit `jobs -l` die meist kürzere Liste der im aktuellen Fenster verwalteten Aufträge anzeigen lassen, was speziell nützlich ist, wenn man die Prozeßnummern von Hintergrundjobs nochmals braucht, um die Prozesse abzubrechen (`kill` Kommando).

```
$sleep 60
CTRL-z
^Z[1] + 13977 Stopped sleep 60
$sleep 90
CTRL-z
^Z[2] + 12698 Stopped sleep 90
$ls -R .. >/dev/null 2>&1 &
$jobs -l
[3] + 14491 Running ls -R .. >/dev/null 2>&1 &
[2] - 12698 Stopped sleep 90
[1] 13977 Stopped sleep 60
$fg
ls -R .. >/dev/null 2>&1
CTRL-z
^Z[3] + 14491 Stopped ls -R .. >/dev/null 2>&1 &
$bg 14491
[3] ls -R .. >/dev/null 2>&1 &
$fg
sleep 90
$fg
sleep 60
$
```

Im Ablauf oben haben wir auch mit `fg` [*job* ...] die angehaltenen Prozesse wieder in den Vordergrund (**f**ore**g**round) gebracht. Umgekehrt kann man im Vordergrund laufende Kommandos mit `bg` [*job* ...] in den Hintergrund (**b**ack**g**round) schicken.

Angehaltene Prozesse erhalten eine Auftragsnummer, die in eckigen Klammern steht ([1], [2], ... oben). Plus und minus sind Kurzbezeichnungen für den letzten und vorletzten angehaltenen Prozeß.

Problematisch ist die Ein-/Ausgabe beim Wechsel von interaktiver zu stapelorientierter (Hintergrund-)Verarbeitung. Sofern wir nicht selbst die Umleitung der Daten besorgen wie im Beispiel, liest ein Hintergrundprozeß von `/dev/null`, bekommt damit sofort *'CTRL-d'* (Eingabeende) und schreibt auch in den großen Biteimer (`/dev/null`). Mit `stty tostop` kann man Hintergrundprozesse beim ersten Ausgabeversuch auch anhalten und sie dann in den Vordergrund holen (`fg`). Analog läßt sich der Leseversuch von der Tastatur abfangen, wenn `monitor on` gesetzt ist; die Meldung auf dem Bildschirm lautet dann `[1] +Stopped (tty input) ...`

Auf die Möglichkeit der *Prozeßsynchronisierung*, d. h. auf Prozesse mit `wait` zu warten, wollen wir hier nicht eingehen. Mehr zur Kommunikation zwischen Prozessen erfahren Sie in Lektion 10.

Zusammenfassung

❖ Wir sahen Anwendungen des Prozeßkonzepts in UNIX bei der Realisierung von Pipes, Kommandoaufrufen mit `&` und periodisch ablaufenden Arbeiten, die das Betriebssystem selbst startet. Mit den Kommandos `nice`, `nohup`, `at` und `ps` können wir Prozesse beeinflussen bzw. deren Status abfragen.

Frage 7

Zur Implementierung der Systemaufgaben und zur Abarbeitung von Benutzeraufträgen gibt es in UNIX zwei sorgfältig getrennte Klassen von Prozessen. Stimmt das?

Frage 8

Die Ausgabe des Kommandos `ps` zeigt den aktuellen Stand der Prozesse

❏ vor dem Aufruf von `ps`,

❏ während der Abarbeitung von `ps`,

❏ nach Beendigung von `ps`.

Frage 9

Das Shell-Skript `mittag` enthalte den Text

```
echo Mahlzeit! ; at noon tomorrow -f mittag
```

und werde einmal mit `at noon -f mittag` aufgerufen. Damit erhält man

❏ genau eine Nachricht „Mahlzeit!“

❏ jeden Mittag eine Nachricht „Mahlzeit!“

❏ eine Fehlermeldung der Art „rekursiver Aufruf von at“

LEKTION 4: Das hierarchische Dateisystem

4.1 Auf dem richtigen Pfad

❖ In diesem Abschnitt geben wir einen Einblick in das baumartige UNIX-Dateisystem und besprechen das Konzept des Pfadnamens. Wir verwenden die Kommandos `pwd` und `ls`.

In den vorhergehenden Lektionen wurden einige Dateien erzeugt, editiert und ausgegeben, etwa mit

- `echo Hallo Leute! >gruss`
- `ed ankuend`
- `tail -3l mehrbest >>nachb`
- `cat nachb`

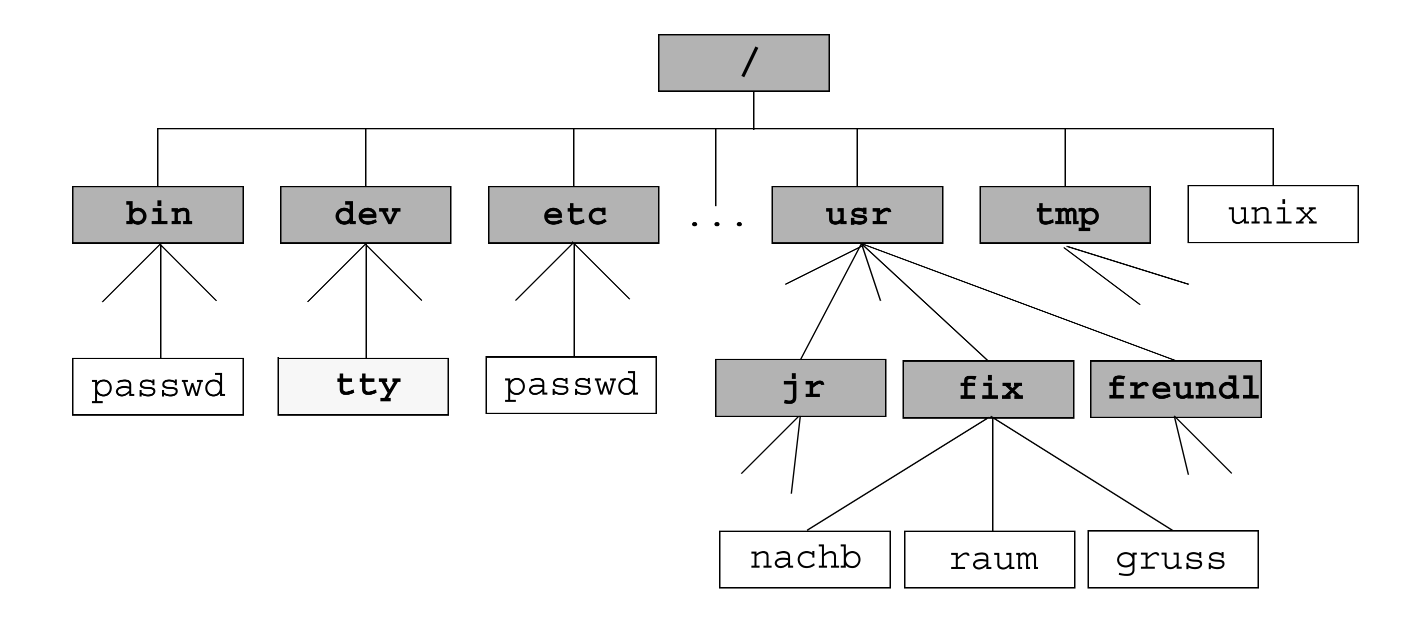
Die in den Kommandos auftretenden Dateien hießen z. B. `gruss`, `ankuend`, `mehrbest` und `nachb`. Man bezeichnet sie als *Normaldateien*. Normaldateien enthalten beliebige

Texte, also z. B. Programmquelltexte, ausführbaren Programmcode oder Schriftstücke für die Druckausgabe.

Allgemein unterscheidet UNIX nur drei Arten von Dateien:

- Normaldateien (normal files),
- Verzeichnisse, Kataloge (directories),
- Gerätedateien (special files).

Alle Dateien werden in einem hierarchischen, also baumartigen Dateisystem zusammengefaßt und organisiert. Die Verzeichnisse bilden die inneren Knoten. Normaldateien, Gerätedateien und leere Verzeichnisse sind die Blätter des Baumes.



Das Bild zeigt einen Ausschnitt aus einem typischen UNIX-Dateisystem.

- `/`, `bin`, `dev`, `etc`, `usr`, `tmp`, `jr`, `fix` und `freundl` sind Verzeichnisse,
- `tty` ist eine Gerätedatei,
- alle anderen, z. B. `nachb`, `raum` und `gruss` von Professor Fix sind Normaldateien.

Einige der oben aufgeführten Verzeichnisse sind in praktisch jedem UNIX-System vorhanden (Standardverzeichnisse). Sie haben typischerweise folgende Inhalte:

- `bin` wichtige ausführbare Programme (binaries), insbesondere Kommandos,
- `dev` Gerätedateien (devices),
- `etc` Systemverwaltungsdateien, z. B. die Paßwortdatei,
- `usr` das Benutzerdateisystem, heute eher `/home` oder `/u`.
- `tmp` temporäre Dateien, die bei jedem Systemstart wieder gelöscht werden.

Jede Datei im Baum läßt sich eindeutig durch die Dateibezeichner auf dem Pfad von der *Wurzel* (`/`) zu der Datei angeben. Die Bezeichner werden dazu ohne Leerzeichen hinterein-

ander geschrieben und mit einem Schrägstrich (/) voneinander getrennt. Eine solche Angabe heißt *voller Pfadname* und beginnt immer mit dem Namen der Wurzel, also mit einem Schrägstrich.

Beispiele für volle Pfadnamen sind

```
/dev/tty
/usr/fix/raum
/bin/passwd
/etc/passwd
/dev
/
/unix
```

Die Angabe von *relativen Pfadnamen* ist eine zweite Möglichkeit, Dateien anzusprechen. Relative Pfadnamen beginnen nicht mit der Wurzel, sondern sind relativ zu dem *aktuellen Verzeichnis*. Statt aktuelles Verzeichnis verwenden wir auch die Begriffe *Arbeitskatalog* und *Arbeitsverzeichnis*. Beispiele:

Relativer Pfadname	angenommenes Arbeitsverzeichnis
`fix/gruss`, `fix`	`/usr`
`passwd`	`/bin`
`etc`, `dev/tty`, `usr/fix/raum`	`/`

Von der Möglichkeit, Dateien relativ zum Arbeitsverzeichnis anzugeben, haben wir bisher schon implizit Gebrauch gemacht. In den Kommandos

```
echo Hallo Leute! >gruss
ed ankuend
tail -3l mehrbest >>nachb
cat nachb
```

waren die Argumente `gruss`, `ankuend`, `mehrbest` und `nachb` relativ zu einem damals nicht genannten Arbeitsverzeichnis. Durch das Kommando `pwd` (**p**rint **w**orking **d**irectory) läßt sich der Pfadname des Arbeitsverzeichnisses ermitteln.

```
$pwd
/usr/fix
$
```

Welche Dateien befinden sich in diesem Verzeichnis?

```
$ls
gruss
mehrbest
nachb
nachb3
raum
```

```
umlauf
$
```

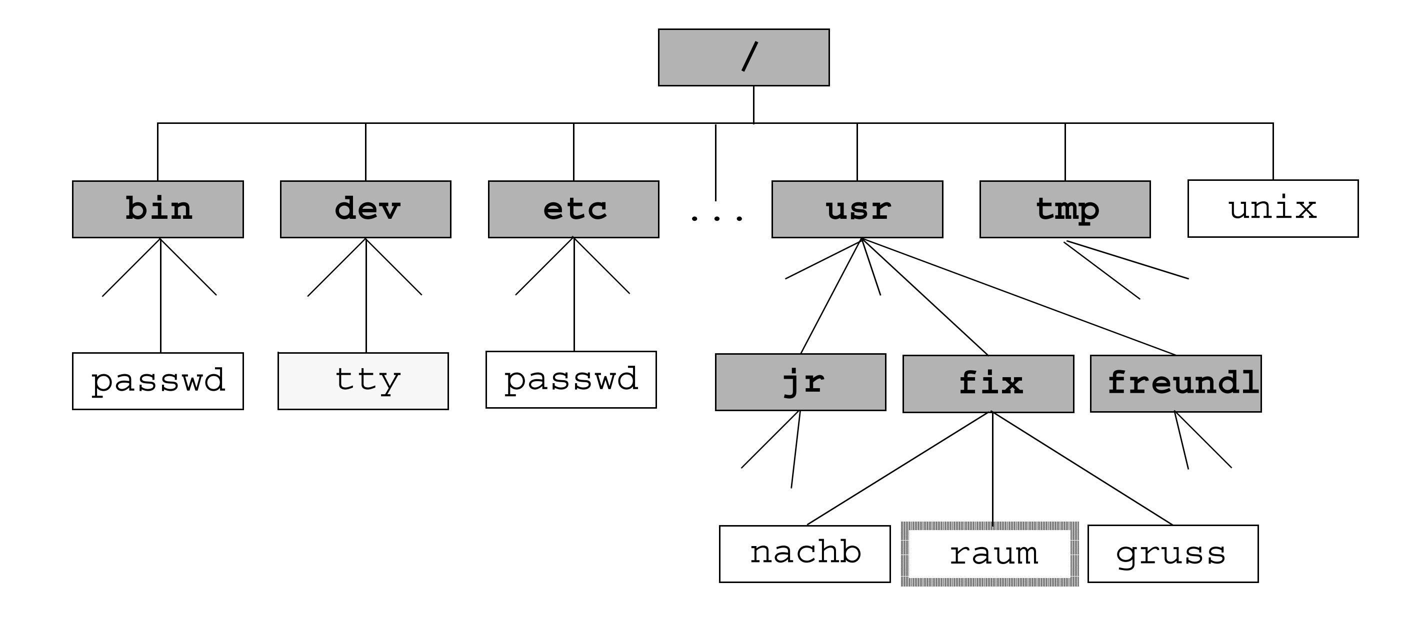
Das Kommando `ls` *katalog* (**li**st content of directory) listet die Einträge eines Katalogs auf. Fehlt das Argument, wird der Inhalt des aktuellen Katalogs gezeigt. Tatsächlich finden sich hier alle unsere bisher erzeugten Dateien wieder.

Zusammenfassung

❖ Wir haben uns einen ersten Eindruck vom hierarchischen Dateisystem mit Verzeichnissen, Normaldateien und Gerätedateien verschafft. Für die Angabe einer Datei unterscheiden wir volle und relative Pfadnamen.

Frage 1

Geben sie den vollen Pfadnamen der markierten Datei `raum` an!



Frage 2

Wie lautet der relative Pfadname der oben markierten Datei, wenn momentan `/usr/fix` das Arbeitsverzeichnis ist?

Frage 3

Nehmen wir an, das Kommando `pwd` liefere `/usr/freundl`. Liefern die Kommandos `ls /usr/freundl` und `ls` dieselbe Ausgabe?

Frage 4

Mögen Sie es knifflig? Bitte sehr!

Das Arbeitsverzeichnis enthalte nur die Einträge `nachb`, `kurs` und `umlauf`. Mit dem Kommando `ls >tricky` werden diese drei Einträge in der Datei `tricky` abgelegt.

- ❏ Das stimmt!
- ❏ Nein - es sind vier Einträge!
- ❏ Das Kommando `ls >tricky` ist prinzipiell falsch.

4.2 In Bewegung

❖ In diesem Abschnitt lernen wir das *Heimatverzeichnis* kennen und bewegen das Arbeitsverzeichnis im Dateibaum. Wir verwenden die Kommandos `cd`, `mkdir` und `rm`.

Beobachten wir Professor Fix wieder bei seiner Arbeit.

```
login: fix
passwd:......
--------------------------------------------
!  MITTEILUNGEN DES SYSTEMVERWALTERS:      !
!  Plattenplatz ist knapp. Bitte unnoetige !
!  Dateien loeschen.                       !
--------------------------------------------
$pwd
/usr/fix
$
```

Unmittelbar nach dem Anmelden hat das Arbeitsverzeichnis von Professor Fix den vollen Pfadnamen `/usr/fix`.

```
$ls
gruss
mehrbest
nachb
nachb3
raum
umlauf
$
```

Die Liste der Einträge enthält die uns bereits bekannten sechs Normaldateien, darunter einige, die nicht mehr benötigt werden. Mit `rm` (**re**mo**ve**) entfernt Professor Fix einigen „Schrott“.

```
$rm gruss mehrbest nachb umlauf
$ls
nachb3
raum
$
```

Mit `mkdir` (**m**a**k**e a **dir**ectory) legt er ein neues Verzeichnis mit Namen `kurs` an.

```
$mkdir kurs
$ls
kurs
nachb3
raum
$
```

Der neue Katalog `kurs` wurde in das Arbeitsverzeichnis eingetragen, aber er ist natürlich noch leer.

```
$ls kurs
$
```

Das leere Verzeichnis `kurs` erkennt man mit `ls` lediglich an der leeren Ausgabeliste. Hier hätte doch eine Meldung der Art „`catalog empty`" nicht geschadet — oder?

Betrachtet man aber die folgende Anwendung — Umlenkung von `ls` in `wc` zum Zählen der Dateieinträge — muß man Bedenken gegen solche Meldungen anmelden.

```
$ls | wc -w
3
$ls kurs | wc -w
0
$
```

Mit der Meldung „`catalog empty`" hätte die letzte Zählung 2 ergeben!

Wir legen nun zwei Dateien im Verzeichnis `kurs` an. Dafür gibt es jedoch kein spezielles Kommando analog zu `mkdir`[1]. Zwei Möglichkeiten sehen wir hier:

```
$echo -n >kurs/lekt1
$cat >kurs/lekt2
CTRL-d
$
```

Die Option `n` bei `echo` unterdrückt das newline-Zeichen, mit dem `echo` üblicherweise seine Ausgabe abschließt, d. h. `lekt1` ist leer. Nach `cat` beenden wir mit 'CTRL-d' die Eingabe, um eine leere Datei zu erhalten.

Bevor wir eine dritte Datei in `kurs` anlegen, ändern wir das Arbeitsverzeichnis, um Schreibarbeit zu sparen. Das Kommando dazu heißt `cd` (**c**hange working **d**irectory) und hat den Zielkatalog als Argument.

```
$cd kurs
$pwd
/usr/fix/kurs
$
```

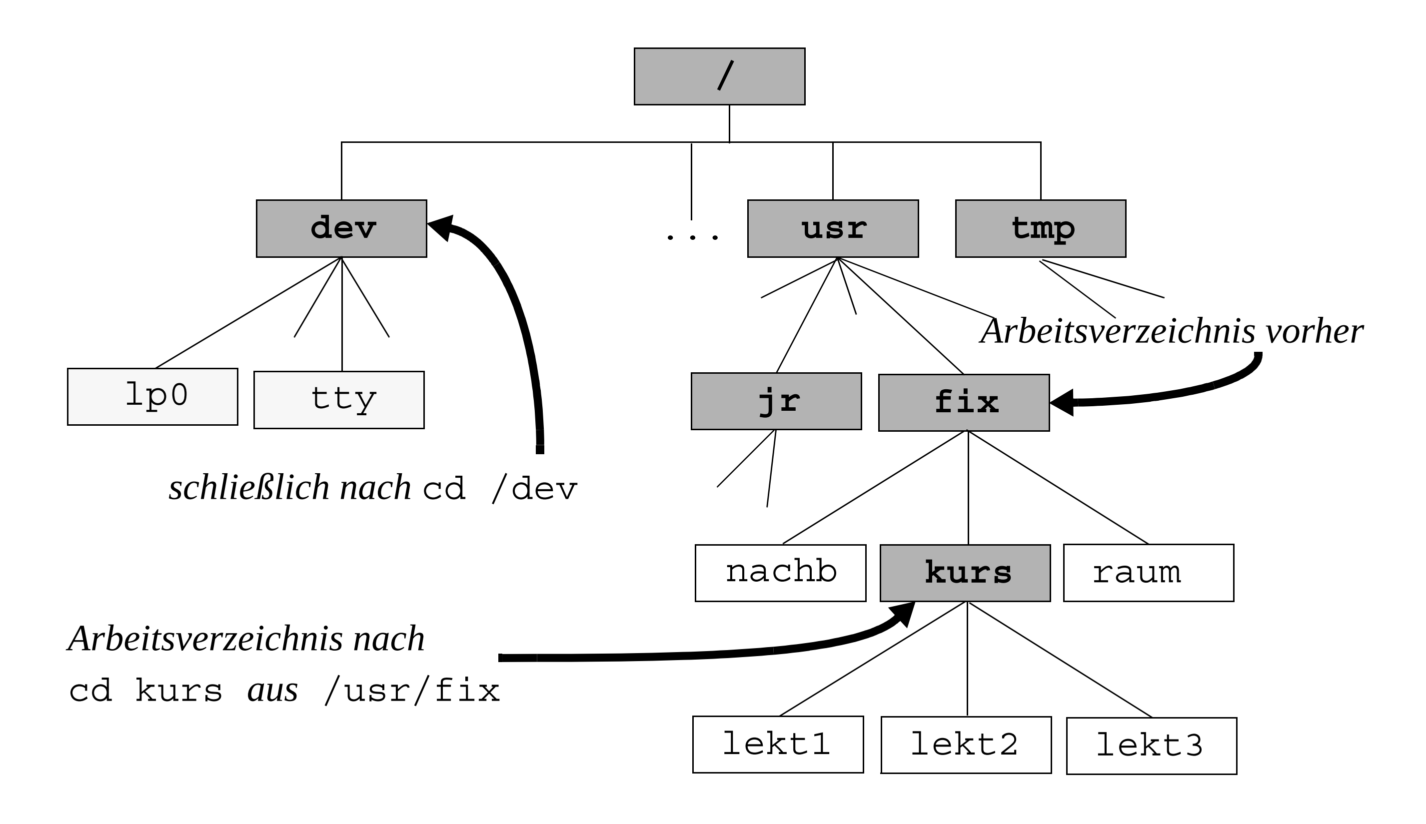
Nun legen wir im neuen Arbeitsverzeichnis, wie bereits angekündigt, nach bekannter Methode noch eine dritte Datei an.

1. Mit dem Kommando `touch` *datei*, das eigentlich zum Setzen der Zugriffszeit einer Datei gedacht ist, läßt sich eine neue, leere Datei anlegen.

```
$echo -n >lekt3
$ls
lekt1
lekt2
lekt3
$
```

Mit cd kann man einen Zeiger von einem Verzeichnis im Dateibaum zu jedem anderen Verzeichnis verschieben. Einschränkungen, die sich aus den Dateirechten ergeben, werden in der übernächsten Lektion behandelt.

Das Verschieben des Zeigers demonstriert das folgende Bild.



Im Bild ist das Arbeitsverzeichnis zunächst /usr/fix. Durch cd kurs versetzen wir den Zeiger von /usr/fix nach /usr/fix/kurs. Von /usr/fix/kurs aus kommen wir mit cd /dev zum Gerätekatalog /dev.

Kann man nicht einen Dateieintrag direkt in ein Verzeichnis schreiben?

```
$cd /usr/fix
$echo lekt4 >>kurs
kurs: cannot create
$
```

Nein, dagegen hat das System Einwände. Zum Schutz der inneren Struktur des Dateisystems, die wir im nächsten Abschnitt kennenlernen, darf niemand, auch nicht der superuser, direkt in Verzeichnisse schreiben.

```
$cd kurs
$pwd
```

```
/usr/fix/kurs
$CTRL-d
```

Was geschieht, wenn sich Professor Fix mittels 'CTRL-d' oder `logout` abmeldet und wieder neu anmeldet? Bleibt das Arbeitsverzeichnis dort, wo es bei der letzten Sitzung war?

```
SIMULIX
Das freundliche Uebungssystem
login: fix
passwd:......
------------------------------------------
!   MITTEILUNGEN DES SYSTEMVERWALTERS:    !
!   neuer C-Compiler im Test              !
------------------------------------------
$pwd
/usr/fix
$
```

Nein, der Zeiger steht wieder auf `/usr/fix`. Nach dem Anmelden ist man immer im login- oder *Heimatverzeichnis*. Ein solches wird für jeden Benutzer vom super-user bestimmt und in der Paßwortdatei (`/etc/passwd`) festgehalten. In den meisten Installationen wählt man Verzeichnisse mit dem login-Namen der Benutzer und trägt sie in das Verzeichnis `/usr`, manchmal auch in einem Verzeichnis `/home`, ein.

Diese Kombination von Login-Name und Login-Verzeichnis nutzen auch fortschrittliche Shells (bash, ksh, csh) für die sog. „*Tilden-Erweiterung*" aus. In einem Ausdruck der Form

~*wort*

wird der Teil von *wort* vor dem ersten Schrägstrich „/" als Login-Name interpretiert und zusammen mit der Tilde „~" durch das Heimatverzeichnis des entsprechenden Benutzers ersetzt.

```
$echo ~fix/hallo
/usr/fix/hallo
$echo ~fax
~fax

$
```

Offensichtlich existiert kein Benutzer `fax`.

In das Heimatverzeichnis wird auch verzweigt, wenn man `cd` ohne Argument aufruft. Es enthält unter anderem eine Datei `.profile` mit individuellen Voreinstellungen des Benutzers, z. B. für spezielle Tastenbelegungen, den Dateibezeichner für die Postablage, usw.

Hier sehen Sie nochmals kurz die Unterschiede zwischen dem Arbeitsverzeichnis und dem Heimatverzeichnis:

Das Arbeitsverzeichnis

- ist anfangs das Heimatverzeichnis,
- kann sich im Dateibaum bewegen, z. B. durch `cd` *zielkatalog*,
- ist einem Prozeß zugeordnet.

Das Heimatverzeichnis

- ist der Einstieg nach dem Anmelden und das Ziel bei `cd` ohne Argument,
- bleibt meist innerhalb einer Sitzung und von Sitzung zu Sitzung unverändert,
- ist einem Teilnehmer zugeordnet.

Zusammenfassung

❖ Wir haben den Unterschied zwischen Heimat- und Arbeitsverzeichnis kennengelernt und mit `cd` das Arbeitsverzeichnis im Baum bewegt. Mit den Kommandos `mkdir` und `rm` wurden Kataloge angelegt und Dateien gelöscht.

Frage 5

Werden `ls` und `cd` ohne Argumente aufgerufen, dann meint man implizit

❑ jeweils das Arbeitsverzeichnis,

❑ jeweils das Heimatverzeichnis,

❑ bei `ls` das Arbeitsverzeichnis, bei `cd` das Heimatverzeichnis,

❑ bei `ls` das Heimatverzeichnis, bei `cd` das Arbeitsverzeichnis.

Frage 6

In welcher Reihenfolge werden bei `ls` (ohne Option) die Einträge ausgegeben?

❑ Sortiert nach dem Datum der letzten Änderung, neuere Dateien zuerst.

❑ Sortiert nach dem Datum des letzten Zugriffs, zuletzt gelesene zuerst.

❑ Sortiert nach Dateinamen.

Frage 7

Eine leere Datei ist in UNIX wirklich leer, d. h. es gibt kein Dateiendezeichen (EOF, EOT oder ähnliches). `echo -n >lekt3; wc lekt3` liefert demnach `0` (Zeilen) `0` (Wörter) `0` (Zeichen) `lekt3`.

Was ergibt `echo >lekt3; wc lekt3` dann?

❑ `0    0    0    lekt3`

❑ `1    0    1    lekt3`

❑ `0 1 1 lekt3`

❑ `1 1 1 lekt3`

❑ `1 1 2 lekt3`

4.3 Ein oder zwei Punkte

❖ In diesem Abschnitt lernen wir mehr über die innere Struktur des Dateisystems kennen. Mit `rm` und `rmdir` entfernen wir Normaldateien und Verzeichnisse.

Professor Fix sitzt immer noch am Terminal, ist aber schon konditionell geschwächt und deshalb leicht orientierungslos. Wo ist er momentan im Dateibaum?

```
$pwd
/usr/fix
$
```

Er ist im Heimatverzeichnis. Welche Einträge enthält es?

```
$ls
kurs
nachb3
raum
$
```

Wurde nicht vorhin behauptet, jedes Heimatverzeichnis enthalte eine Systemdatei `.profile` mit Anpassungsparametern? Die Option `a` (alles zeigen) bei `ls` bringt es an den Tag.

```
$ls -a
.
..
.profile
kurs
nachb3
raum
$
```

UNIX versteckt Dateien, die mit einem Punkt beginnen[1]. Nach dem Namen für das Wurzelverzeichnis (/) sehen wir nun zwei weitere „merkwürdige“ Namen, nämlich *Punkt* (**.**) und *PunktPunkt* (**..**). Was verbirgt sich hinter *Punkt*?

```
$ls .
kurs
nachb3
```

1. Im Heimatverzeichnis des Verfassers sind es inzwischen 41, von .Softlpex.ras, .Softstatic.ras, .WWW über .Xkeyboard und .bash_history bis hin zu .xsession-errors — vielleicht die Überstrapazierung einer an sich guten Idee?

```
raum
$
```

Das kommt Professor Fix bekannt vor — dasselbe hat er mit `ls` ohne Argument erhalten. *Punkt* ist also der eingetragene *Name des gegenwärtigen Katalogs*. Professor Fix bestätigt seine Vermutung, indem er auch in `kurs` nachsieht.

```
$ls -a kurs
.
..
lekt1
lekt2
lekt3
$
```

Auch hier sind die Einträge *Punkt* und *PunktPunkt* im Katalog, und *Punkt* liefert wieder den Katalog selbst, wie man im folgenden sieht.

```
$ls kurs/.
lekt1
lekt2
lekt3
$
```

Professor Fix bewegt das Arbeitsverzeichnis nach `kurs`. Welches Verzeichnis ist *Punkt-Punkt*?

```
$cd kurs
$ls ..
kurs
nachb3
raum
$
```

Offensichtlich ist *PunktPunkt* das Vorgängerverzeichnis `/usr/fix`, von wo Professor Fix gerade gekommen ist. Mit `ls` haben wir uns seine Einträge angeschaut.

Tatsächlich stehen *Punkt* und *PunktPunkt* auch so textuell im Katalog, wie man sich mit einem oktalen Speicherauszug (`od -c`) des Katalogs `kurs` ansehen kann:

```
$od -c ../kurs
0000000 Ç    . \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000020   \n .   . \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040 Ç    l  e  k  t  1 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000060 Ç    l  e  k  t  2 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000100 Ç    l  e  k  t  3 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000120
$
```

Das Kommando `cd ../..` müßte uns somit zwei Stufen höher im Baum bringen,

```
$cd ../.. ; pwd
```

```
/usr
$
```

aber

```
$cd .. ; pwd
/
$cd ..
$pwd
/
$
```

über die Wurzel hinaus kann es schlecht gehen.

Die Einträge *Punkt* und *PunktPunkt* sind in jedem Verzeichnis vorhanden, auch in einem sonst leeren Verzeichnis.

```
$cd /usr/fix
$mkdir kurs2
$ls kurs2
$ls -a kurs2
.
..
$
```

Keine Einträge in `kurs2`, außer natürlich den versteckten Einträgen. Professor Fix möchte das Verzeichnis `kurs2` gleich wieder löschen.

```
$rm kurs2
rm: kurs2 directory
$rmdir kurs2
$
```

Mit `rm` geht dies nicht. Zum Löschen von Katalogen hält UNIX das Kommando `rmdir` (**rem**ove a **dir**ectory) bereit. UNIX erledigt dies, ohne Fragen zu stellen. Professor Fix entschließt sich, auch den Katalog `kurs` zu löschen.

```
$rmdir kurs
rmdir: kurs not empty
$
```

Jetzt funktioniert `rmdir` nicht! Der Grund: Der Katalog `kurs` ist nicht leer! Versehentliches Löschen wird hierdurch verhindert.

```
$cd kurs
$rm lekt?
$
```

Mit der Abkürzung `lekt?` für `lekt1`, `lekt2` und `lekt3` löscht Professor Fix die Einträge in `kurs`. Das Fragezeichen steht für ein einzelnes Zeichen und wird von der Shell passend für die Einträge des Arbeitsverzeichnisses substituiert. Mehr zu diesem Metazeichen finden Sie später bei der Erläuterung der Shell.

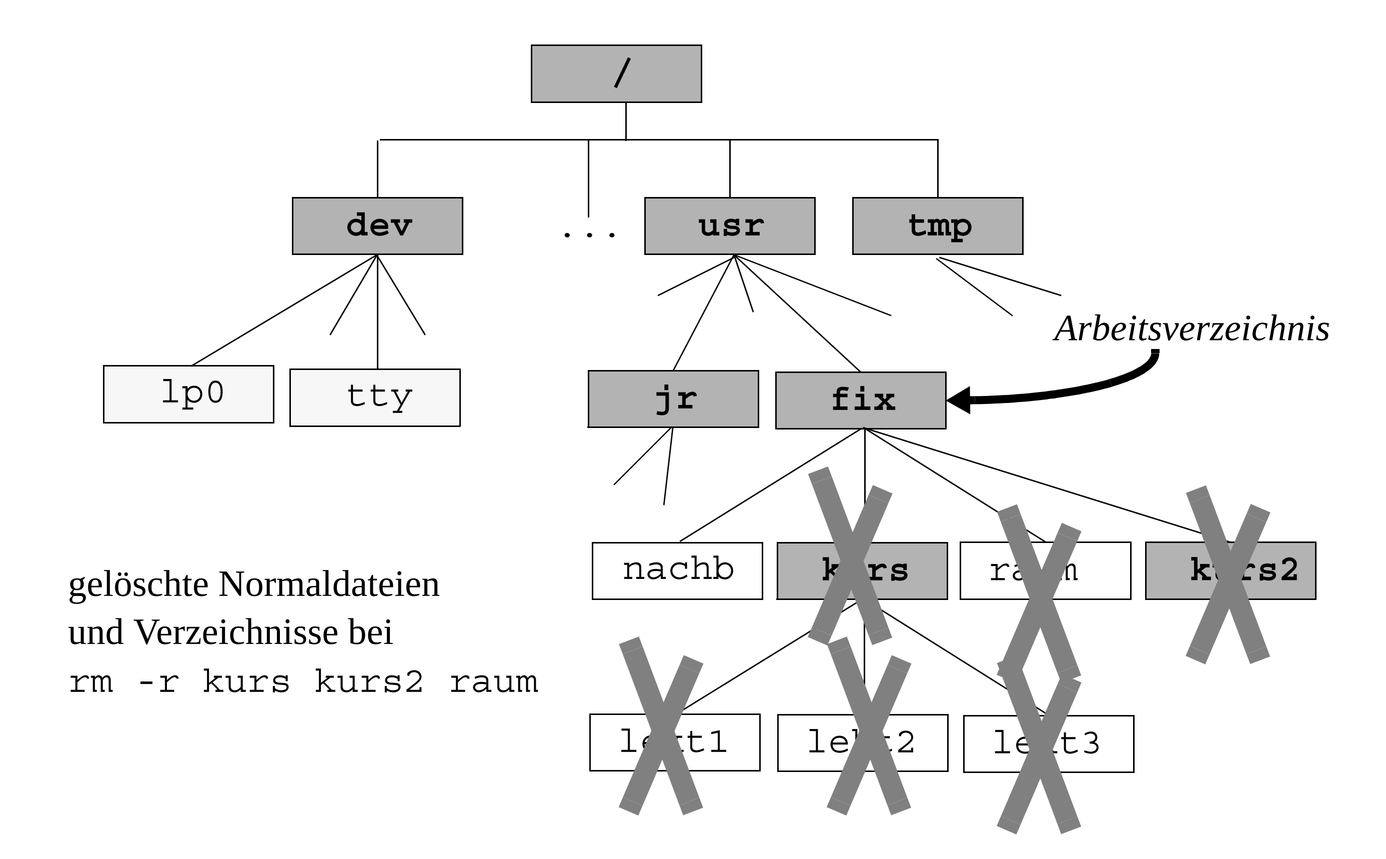
```
$rmdir kurs
rmdir: kurs non-existent
$
```

Jetzt hat `rmdir kurs` nicht funktioniert, weil Professor Fix in `kurs` selbst noch stand, statt in dessen Vorgänger. Auch durch Angabe des vollen Pfadnamens geht es nicht — wahrhaft eine zähe Angelegenheit!

```
$rmdir /usr/fix/kurs
rmdir: cannot remove current directory
$cd ..
$rmdir kurs
$
```

Na endlich! Wer hätte daran noch geglaubt.

Schneller wäre das Löschen mit `rm -r kurs kurs2` gegangen. Durch die Option `-r` (rekursiv) werden alle Einträge in den angegebenen Verzeichnissen `kurs` und `kurs2` sowie anschließend die Verzeichnisse selbst gelöscht. Dies zeigen das nächste Bild. Natürlich können als Argumente von `rm -r` auch noch Normaldateien angegeben werden.



Schnell heißt aber auch gefährlich!

Da UNIX bei der Ausführung von `rm` ohne die Option `i` (interaktiv) keine Rückfragen stellt, können ganze Unterbäume verschwinden, ohne daß der Besitzer, außer einem unschuldigen `$` nach Eintritt der Katastrophe, etwas sieht.

Aus reiner Menschenfreundlichkeit weigert sich UNIX aber, bei `rm -r` den Eintrag *PunktPunkt* zu löschen - das hat schon vielen Dateisystemen bei schlimmen Anschlägen der Art `rm -r .` das Leben gerettet.

Zusammenfassung

❖ Wir haben über die Option `-a` bei `ls` die versteckten Dateien *Punkt* (`.`) und *PunktPunkt* (`..`) kennengelernt, wobei *Punkt* das Verzeichnis selbst und *PunktPunkt* den Vorgänger bezeichnet. Mit `rmdir` und `rm -r` haben wir Verzeichnisse und deren Inhalte gelöscht.

Frage 8

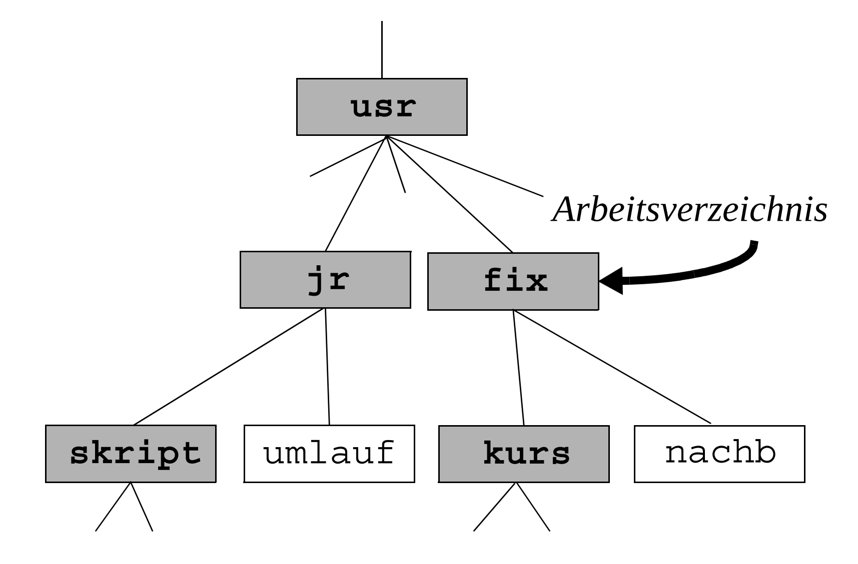
Man betrachte den Dateibaum unten. Mit welchen Aufrufen von `cd` kommt Professor Fix in `jr`'s Katalog `skript`? (Vorausgesetzt, die Dateirechte erlauben es.)

❏ `cd /usr/jr/skript`

❏ `cd .usr/jr/skript`

❏ `cd ../jr/skript`

❏ `cd jr/skript`



Frage 9

Welches der folgenden Kommandos ist ausgesprochen sinnlos?

❏ `cd`

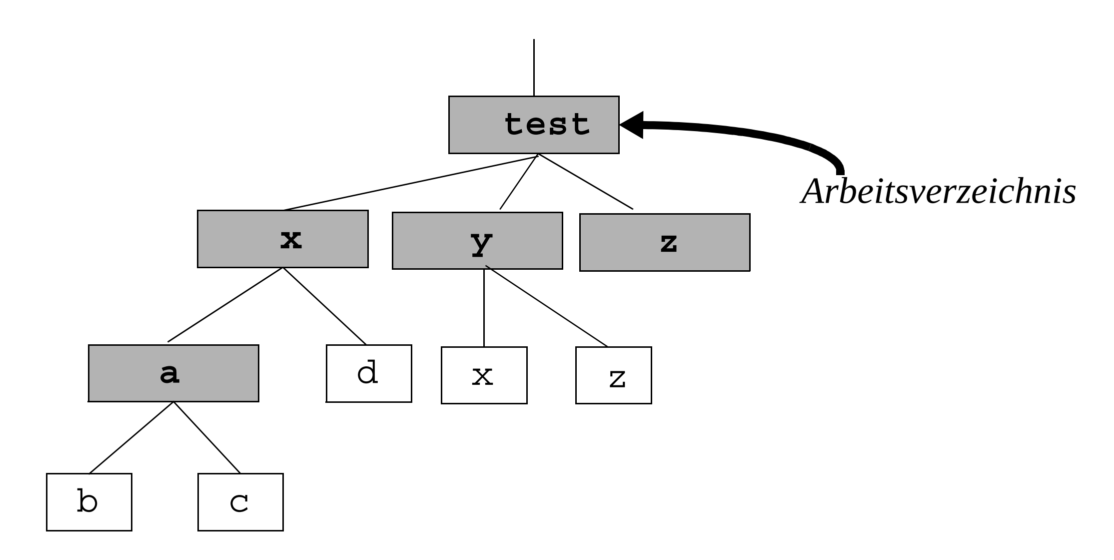
❏ `cd .`

❏ `ls`

❏ `ls .`

Frage 10

Wieviele Dateien (also Kataloge und Normaldateien) werden aus dem folgenden Dateibaum durch das Kommando `rm -r x z` entfernt?



Frage 11

In der Bash- und Korn-Shell kann man mit `cd -` den letzten Verzeichniswechsel durch einen `cd`-Aufruf wieder rückgängig machen. War ein Anwender im Verzeichnis `test` aus dem Dateibaum in Frage 10 und hat er zuletzt `cd x/a` ausgeführt, dann kommt er mit `cd -` wieder

❑ nach `a`.

❑ nach `x`.

❑ nach `test`.

LEKTION 5: Dateien kopieren und verlagern

5.1 Kopie genügt

❖ In diesem Abschnitt kopieren wir Dateien mit `cp`, ändern eine Datei innerhalb des Editors `ed` mit dem Editorkommando `s` und vergleichen Dateien mit Hilfe des Kommandos `diff`.

Professor Fix kündigt die Veranstaltungen für den Sommer an.

```
$cat >bs
Ankuendigungen fuer das SS
Betriebssysteme (2+2)
fuer Hoerer im Hauptstudium.
Zeit: Di 10-12
Ort: HS 100
$
```

Er gibt den Text mit `cat` direkt ein und schließt die Eingabe mit ‘CTRL-d’ ab. Für die Veranstaltung „Datenorganisation“ kopiert er zunächst mit `cp` (**co**p**y**) die Datei `bs` in die Datei `do`. Das Original bleibt dabei unangetastet.

```
$cp bs do
$ls
bs
do
nachb3
raum
$
```

Die Kopie wird, wie zuvor schon `bs`, im Arbeitsverzeichnis eingetragen. Aber Vorsicht!

Hätte es eine nicht schreibgeschützte Datei `do` dort bereits gegeben, wäre ihr alter Inhalt gelöscht worden. Falls es Ihnen ein Trost ist, bei `cat <bs >do` wäre es uns nicht besser ergangen.

Die Dateien `bs` und `do` haben bis jetzt noch denselben Inhalt. Mit dem Kommando `diff` (**dif**ferential **f**ile comparator, Dateivergleich) kann man dies prüfen. Es sagt uns, welche Zeilen in der Datei `bs` geändert werden müßten, um sie in Übereinstimmung mit `do` zu bringen.

```
$diff bs do
$
```

Momentan erkennt `diff` keine Unterschiede, deshalb erscheint nur das Bereitzeichen als Antwort.

Jetzt editiert[1] Professor Fix die Datei `do`. Mit `1,$p` (**p**rint Zeile 1 bis $ = letzte Zeile) schaut er sich `do` nochmals an.

```
$ed do
108
1,$p
Ankuendigungen fuer das SS
Betriebssysteme (2+2)
fuer Hoerer im Hauptstudium.
Zeit: Di 10-12
Ort: HS 100
```

Mit dem Editor-Kommando `s` (**s**ubstitute) ersetzt er in den Zeilen 2 und 4 die zu ändernden Zeichen. Mit `a` fügt er eine neue Zeile an.

```
2s/Betriebssysteme/Datenorganisation
Datenorganisation (2+2)
2s/2+2/3+1
Datenorganisation (3+1)
4s/Di 10-12/Do 14-16
Zeit: Do 14-16
5a
gez. Fix
```

1. hier nochmals mit `ed` um die Substitution zu üben, wie man sie im Streameditor `sed` für kleine Batch-Jobs (Shell-Skripte) und in der Kommandozeile von vi braucht. Ansonsten ist generell ein komfortablerer Editor zu empfehlen.

Nachdem Professor Fix die modifizierte Datei `do` zurückgeschrieben hat, beendet er den Editor und schaut sich `do` nochmals an.

```
.
w
120
q
$cat do
Ankuendigungen fuer das SS
Datenorganisation (3+1)
fuer Hoerer im Hauptstudium.
Zeit: Do 14-16
Ort: HS 100
gez. Fix
$
```

Jetzt zeigt uns `diff` die Unterschiede zwischen den beiden Dateien `bs` und `do`.

```
$diff bs do
2c2
< Betriebssysteme (2+2)
> Datenorganisation (3+1)
4c4
< Zeit: Di 10-12
> Zeit: Do 14-16
5a6
> gez. Fix
$
```

Dabei bedeutet `c` zu ändernde, `d` zu löschende und `a` anzufügende Zeilen, damit `bs` zu `do` wird.

Eigentlich hätte man die Dateien `bs` und `do` in ein eigenes Unterverzeichnis `aushang` eintragen können. Dazu kann man die zweite Form des `cp` Kommandos

`cp` *datei1* [*datei2* ...] *katalog*

verwenden, bei der die Kopien unter ihrem alten Namen in den angegebenen Katalog eingetragen werden.

```
$mkdir aushang
$cp bs do aushang
$rm bs do
$
```

Dies ist aber eine umständliche und aufwendige Lösung, wenn man keine echte Kopie will. Im nächsten Abschnitt zeigen wir, wie es in solchen Fällen mit `mv` (**mov**e and rename) eleganter geht.

Weil wir wissen, daß Sie uns nicht verraten, erzählen wir Ihnen auch, daß man mit `cp` *datei* `/dev/lp0` auf den meisten Anlagen eine Datei direkt auf dem Zeilendrucker (**l**ine **p**rin-

ter) ausgeben kann. Dies ist möglich, da Geräte wie Dateien behandelt werden. Allerdings wird dabei der Druck-Spooler aus `lpr` umgangen, eine gefährliche Sache im Mehrbenutzerbetrieb.

Eine wichtige Anwendung von `cp` ist natürlich auch die *Dateisicherung*. Ein Kommando der Form

```
cp index kap? backup
```

zusammen mit einer periodischen Datensicherung des Katalogs `backup` auf andere Datenträger (z. B. Diskette, DAT-Kassette, wiederbeschreibbare CD, usw.) und geeignete Dateisperren (siehe Lektion 6) haben schon viele Tränen erspart.

`locate` und `find`: Wer sucht der findet

Mit der Zeit sammeln sich im Heimatverzeichnis und den Verzeichnissen darunter eine enorme Anzahl von Dateien an. Je größer die Sammlung (und je chaotischer die Ordnung), desto öfter kommen Fragen wie „Wo war denn gleich die Datei, in der ich die UNIX-Vorlesung angekündigt habe?“ Für dieses Problem gibt es das Kommando

`find` *pfadnamen bedingungen*,

mit dem man Verzeichnisstrukturen nach anzugebenden Kriterien durchsuchen kann. Im obigen Beispiel würde durch Aufruf von

```
find /usr/fix -name 'UNIX*' -print
```

das Verzeichnis `/usr/fix` und alle Unterverzeichnisse rekursiv nach einer Datei durchsucht, deren Namen (`-name`) auf das Muster `'UNIX*'` paßt. Gefundene Dateien oder Verzeichnisse werden in diesem Fall ausgegeben (`-print`).

Andere Suchkriterien wären etwa Dateitypen und -größen, Benutzerkennungen und -rechte, sowie Zugriffszeiten, die in einem Boole’schen Ausdruck kombinierbar sind. Für die gefundenen Dateinamen können dann Aktionen ausgeführt werden. Neben der Ausgabe der Fundstellen sind noch beliebige andere Shell-Kommandos möglich, z. B. die Durchsuchung (`grep`) der gefundenen Dateien nach einer Zeichenkette, unten etwa das Wort 'Vorlesung', wobei { } durch den Namen der aktuellen Datei ersetzt wird.

```
find /usr/fix -name 'UNIX*' -exec grep "Vorlesung" {} \; -print
```

Ein Nachteil von `find` ist, daß die Suche im Dateisystem relativ zeitaufwendig ist, da die Verzeichnisse über die Festplatte verteilt gespeichert sind. Für die häufigste Anwendung von `find`, der Suche nach Dateinamen, gibt es von GNU das Kommando `locate`. Dieses sucht die Dateinamen in einer Datenbank, die eine Momentaufnahme der gesamten Verzeichnishierarchie darstellt. Wenn die Datenbank nur oft genug auf den neusten Stand gebracht wird (z. B. zweimal täglich), liefert `locate` sehr schnell ein hinreichend zuverlässiges Ergebnis.

Zusammenfassung

❖ Wir haben `cp` in zwei Formen zum Kopieren kennengelernt, mit `s/`*muster*`/`*text*`/` im Editor `ed` die Zeichenkette *muster* durch die Zeichenkette *text* ersetzt und mit `diff` die Unterschiede zweier Dateien ermittelt.

Frage 1

Liefert der Vergleich von `bs` und `do` mittels `diff` bei den beiden Kommandos

```
diff bs do
diff do bs
```

die gleiche Ausgabe?

❏ Ja, immer.

❏ Nur wenn `do` gleich `bs` ist.

❏ Nein, nie.

Frage 2

Mit `1,$p` gibt man im `ed` den ganzen Editorpuffer aus. Welche Änderung des Musters „Unix“ zu „UNIX“ bewirkt wohl `1,$s/Unix/UNIX/g` ?

❏ Änderung an der ersten gefundenen Stelle.

❏ Änderung an allen Stellen.

❏ Änderung nur in der ersten und letzten Zeile.

Frage 3

Mit `mkdir aushang` wurde ein Verzeichnis angelegt, in das `bs` und `do` kopiert werden sollen. `ls` liefert `aushang`, `bs`, `do`, `nachb3` und `raum`. Welche der fünf Alternativen **versagt**?

❏ `cp bs do aushang`

❏ `cd aushang; cp bs do .`

❏ `cd aushang; cp ../bs ../do .`

❏ `cp bs aushang/bs; cp do aushang/do`

❏ `cd aushang; cp ../bs bs; cp ../do do`

5.2 Schiebung

❖ In diesem Abschnitt lernen wir das Kommando `mv` zur Namensänderung und Verschiebung von Dateien kennen. Mit `write` schreiben wir auf andere Terminals.

Professor Fix hat gerade die Aushänge gedruckt, da kommt eine Botschaft für ihn. Mit dem Kommando `write fix` schreibt jemand auf sein Terminal.

```
$cat bs do | lpr
$Message from freundl (tty3) ...
Neue Studienordnung fertig. (o)
$
```

Mit `write freundl` antwortet Professor Fix der Sekretärin. Dadurch wird ein Dialog von Terminal zu Terminal möglich,

```
$write freundl
Wo steht sie? (o)
Im Katalog /usr/freundl/sto! (o)
Danke - ich kopiere sie mir. (oo)
Alles klar, Herr Fix. (oo)
<EOT>
$
```

der wie üblich mit 'CTRL-d' beendet wird. Für die Abwicklung des Dialogs hat sich ein informelles Protokoll mit `(o)` (over) für die Abgabe des Senderechts und `(oo)` (over and out) als Signal für den Abbruch der Kommunikation eingebürgert, ähnlich wie im Sprechfunk, das natürlich nicht vom Betriebssystem oder der Shell „verstanden" wird.

Nun holt sich Professor Fix die Dateien von Frau Freundlich. Er benutzt dabei wieder die Form `cp` *d1* [*d2* ...] *katalog* des `cp` Kommandos, mit der die Dateien *d1*, *d2*, ... unter ihrem alten Namen in den angegebenen Katalog kopiert werden.

```
$cd /usr/freundl/sto
$ls
status
sto.alt
sto.neu
$cat status
sto.neu ist Ueberarbeitung von sto.alt
Diktat von jr, gez. Freundl
$cp status sto.alt st.neu /usr/fix
$cd /usr/fix
$
```

Kürzer wäre es mit `cp * /usr/fix` gegangen, wobei das Metazeichen Stern (`*`) für alle Einträge im Arbeitsverzeichnis steht (näheres dazu später). Auch möglich wäre gewesen, mit `cp` und einer der Optionen `r` oder `R` (rekursiv) das ganze Verzeichnis `sto` samt Dateien in das Heimatverzeichnis von Professor Fix zu kopieren, wie unten gezeigt.

```
cd /usr/freundl
cp -R sto /usr/fix
```

Kehren wir zur ersten Lösung zurück. Welche Dateien stehen jetzt im Arbeitsverzeichnis? Professor Fix bereitet die Dateiliste mit dem Kommando `pr` (**pr**int files) durch die Option `3t` dreispaltig ohne Kopfzeile (Seitentitel) auf.

```
$ls | pr -3t
bs            raum             sto.neu
do            status
nachb3        sto.alt
$
```

Professor Fix möchte lieber andere Namen für die neuen Dateien. Er ändert Sie mit `mv` (**mov**e or rename files and directories) und verwendet dabei zur Abwechslung auch mal Großbuchstaben.

```
$cp sto.neu StO97.3
$mv sto.neu StO97.2
$mv sto.alt StO97.1
$
```

Die zwei `mv` Kommandos benutzten die Form `mv` *datei1 datei2* und ändern damit den Dateinamen von *datei1* zu *datei2*, ohne die Datei zu kopieren. Eine existierende *datei2* wird dabei allerdings überschrieben.

```
$rm sto.neu
rm: sto.neu non-existent
$
```

Tatsächlich ist keine Datei mehr mit dem Namen `sto.neu` vorhanden. Die beiden folgenden Bilder zeigen nochmals den Unterschied zwischen Kopieren und Umbenennen.

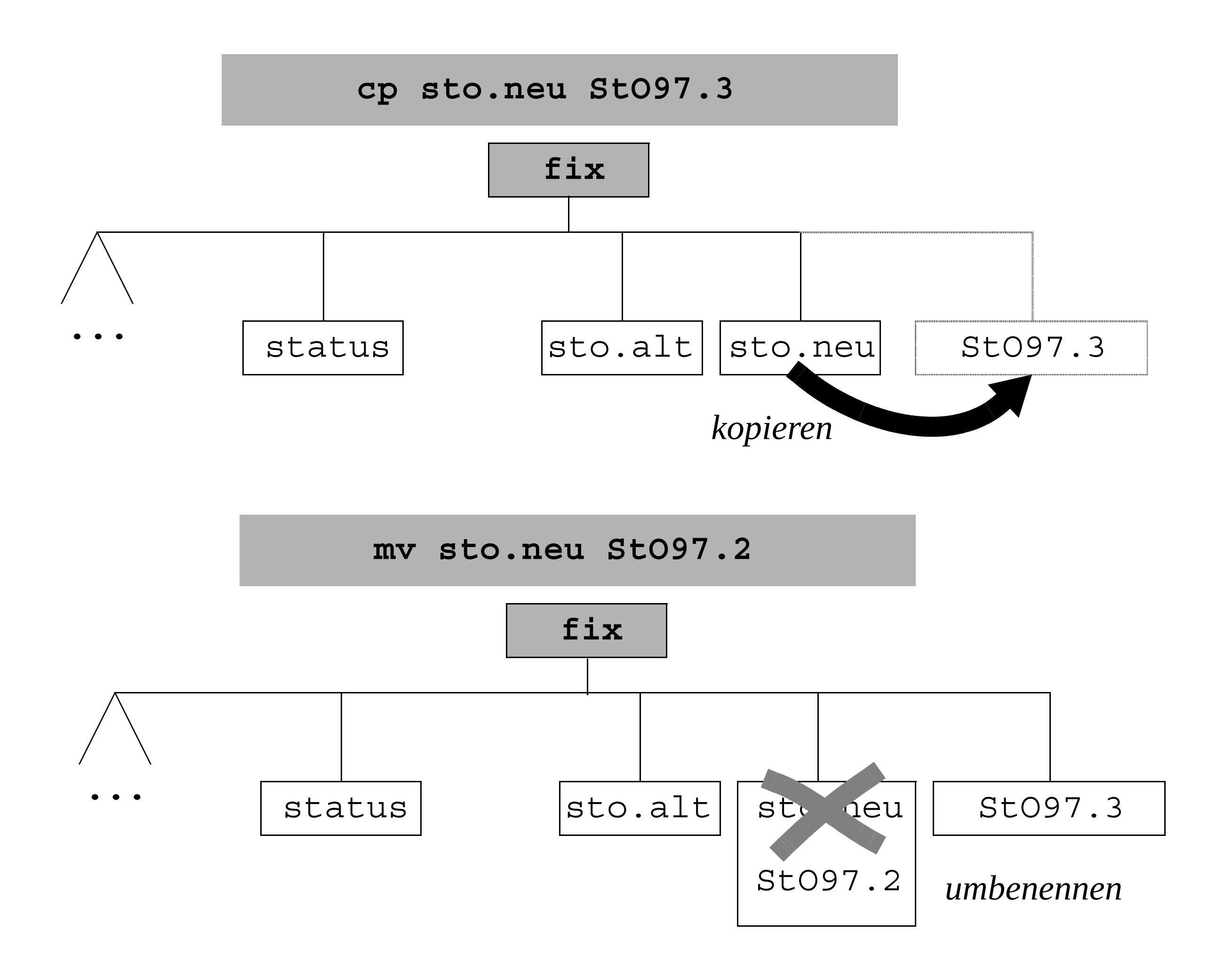
Eigentlich sollten die vier Dateien in ein gesondertes Unterverzeichnis für Studienordnungsangelegenheiten. Nichts leichter als das.

```
$mkdir sto
$mv status StO97.? sto
$ls | pr -4t
aushang       nachb3           raum             sto
$cd sto; ls
StO97.1
StO97.2
StO97.3
status
$
```

Das Fragezeichen wird von der Shell wieder durch die Einzelzeichen `1`, `2` und `3` ersetzt. In dieser zweiten Form

`mv` *datei ... katalog*

wird wie bei `cp` der alte Name für den Eintrag in den Katalog genommen.



Daneben kann man `mv`, anders als `cp` ohne `R` Option, auch mit Verzeichnissen als Argumenten aufrufen, d. h. auch Namen von Verzeichnissen sind änderbar.

```
$cd ..
$mv sto studord
$ls | pr -2t
aushang                 raum
nachb3                  studord
$
```

Neuere UNIX-Versionen erlauben auch das Verlagern ganzer Teilbäume durch Angabe voller Pfadnamen für das Ziel.

```
$Message from jr (tty1)...
Mittagessen? (o)
write jr
Mensa? (o)
Ja, ich bin in Eile (o)
Gebongt (oo)
Bin schon unterwegs (oo)
<EOT>
CTRL-d
$
```

Zeit aufzuhören? Natürlich erst nach den Fragen!

Zusammenfassung

❖ Wir haben Dateien mit `mv` umbenannt und in andere Verzeichnisse verlagert. Mit `write` wurde ein Dialog zwischen zwei Terminals geführt.

Frage 4

Wie hätte man die Befehlsfolge

```
$cp do bs aushang
$rm do bs
```

besser und weniger aufwendig durch ein Kommando ersetzen können?

Frage 5

Neuere UNIX-Versionen erlauben das Verlagern von Dateibäumen mit `mv`. Was bewirkt dann `mv fix /usr/fix/sto`, wenn `/usr` der aktuelle Katalog ist?

❏ Das kann nicht gehen.

❏ Das Kommando verlagert `fix` in den eigenen Dateibaum.

❏ Das Kommando verlagert alle Dateien im aktuellen Katalog nach `/usr/fix/sto`.

Frage 6

Hätte Professor Fix statt `write` auch unterschiedslos `mail` verwenden können?

❏ Ja, kein Unterschied.

❏ Geringer Unterschied (z. B. Protokoll `(o)` und `(oo)`).

❏ Großer Unterschied (z. B. on-line/off-line Kommunikation).

❏ Nicht vergleichbar (z. B. `write` sendet, `mail` empfängt).

Frage 7

Weil ein gutes Betriebssystem vieles mitmacht, hat Professor Fix wieder verrückte Ideen. Wie lautet die ausgelassenen Zeile an seinem Terminal?

```
$ ?
Message from fix (tty5) ...
```

5.3 Gelinkt

❖ In diesem Abschnitt lernen wir das Konzept der i-Knoten kennen und verwenden das `ln` Kommando.

Kopieren geht über studieren, behaupten böse Zungen. Das gilt auch für Dateien auf Rechenanlagen. Nehmen wir die überarbeitete Studienordnung aus dem vorhergehenden Abschnitt. Sie muß allen zugänglich gemacht werden. Dazu könnten wir

- den Text ausdrucken und damit Personal, Kopierer und Umwelt mit Papierverarbeitung belasten,
- eine „elektronische“ Kopie in alle Heimtatverzeichnisse legen,
- allen über `mail` nur den Aufenthaltsort der Datei mitteilen.

Die erste Variante ist schlecht, die zweite schon besser und die dritte hätten Sie natürlich sowieso vorgeschlagen.

```
$mail
From dekan Tue Feb 4 14:20 GMT 1997
Neue Studienordnung StO97 liegt aus in
/usr/fbinfo
Gruss D.Boss
d
$
```

Diese Lösung hat aber auch einige Nachteile:

- Man muß sich den genauen (Pfad-) Namen merken.
- Das Wechseln in fremde Dateibäume ist lästig.
- Ändert jemand den (Pfad-) Namen der Datei, müssen alle anderen suchen.
- Macht man sich eine Kopie der Datei und wird später das Original geändert, hat man nicht mehr den neusten Stand.

Deshalb sieht UNIX neben `cp` und `mv` das Kommando `ln` (make a **lin**k, einen Zeiger setzten) vor. Damit läßt sich ein physisches Exemplar einer Datei unter mehreren Namen in verschiedenen Verzeichnissen eintragen. Die Syntax lautet `ln` *name1 name2*.

Für die Datei *name1* wird zusätzlich der Name *name2* vergeben. Ist *name2* ein Katalog, wird die letzte Komponente des Pfadnamens von *name1* in den Katalog eingetragen. Diese Form ist aber nicht auf allen Systemen erlaubt.

Wir sehen nach, was in `/usr/fbinfo` und im aktuellen Katalog eingetragen ist.

```
$ls /usr/fbinfo | pr -3t
StO95 StO97 wahlen
$ls | pr -3t
aushang     nachb3          raum
$
```

Nun wenden wir das Kommando `ln` an, um die Studienordnung auch im eigenen Katalog zur Verfügung zu haben.

```
$ln /usr/fbinfo/StO97 .
$ln /usr/fbinfo/StO95 StO-alt
```

```
$ls | pr -3t
StO-alt      aushang        raum
StO97        nachb3
$
```

Die neuen Namen StO97 und StO-alt im Arbeitsverzeichnis werden gleichgesetzt mit StO97 und StO95 aus /usr/fbinfo. Anders als bei mv bleiben StO97 und StO95 in /usr/fbinfo bestehen.

Das ln Kommando läßt sich leicht verstehen, wenn man das Organisationsprinzip des Dateisystems betrachtet. Weil UNIX ein offenes Betriebssystem ist, läßt es sich gut in die Karten schauen. Der Zugang ist die Option i (zeige i-Nummer) des Kommandos ls.

```
$ls -i | pr -2t
715 StO-alt         517 nachb3
710 StO97           495 raum
693 aushang
```

Jede *i-Nummer* entspricht einem sogenannten *i-Knoten* (*i-node*, häufiger schon *inode*). Diese stellen die Verbindung zwischen dem logischen und dem physischen Dateisystem her. Die Bezeichnung i-Nummer deutet an, daß es sich um einen Index in eine Knotentabelle handelt. Über i-Knoten später mehr.

```
$ls -ai aushang
693 .
362 ..
651 bs
655 do
$
```

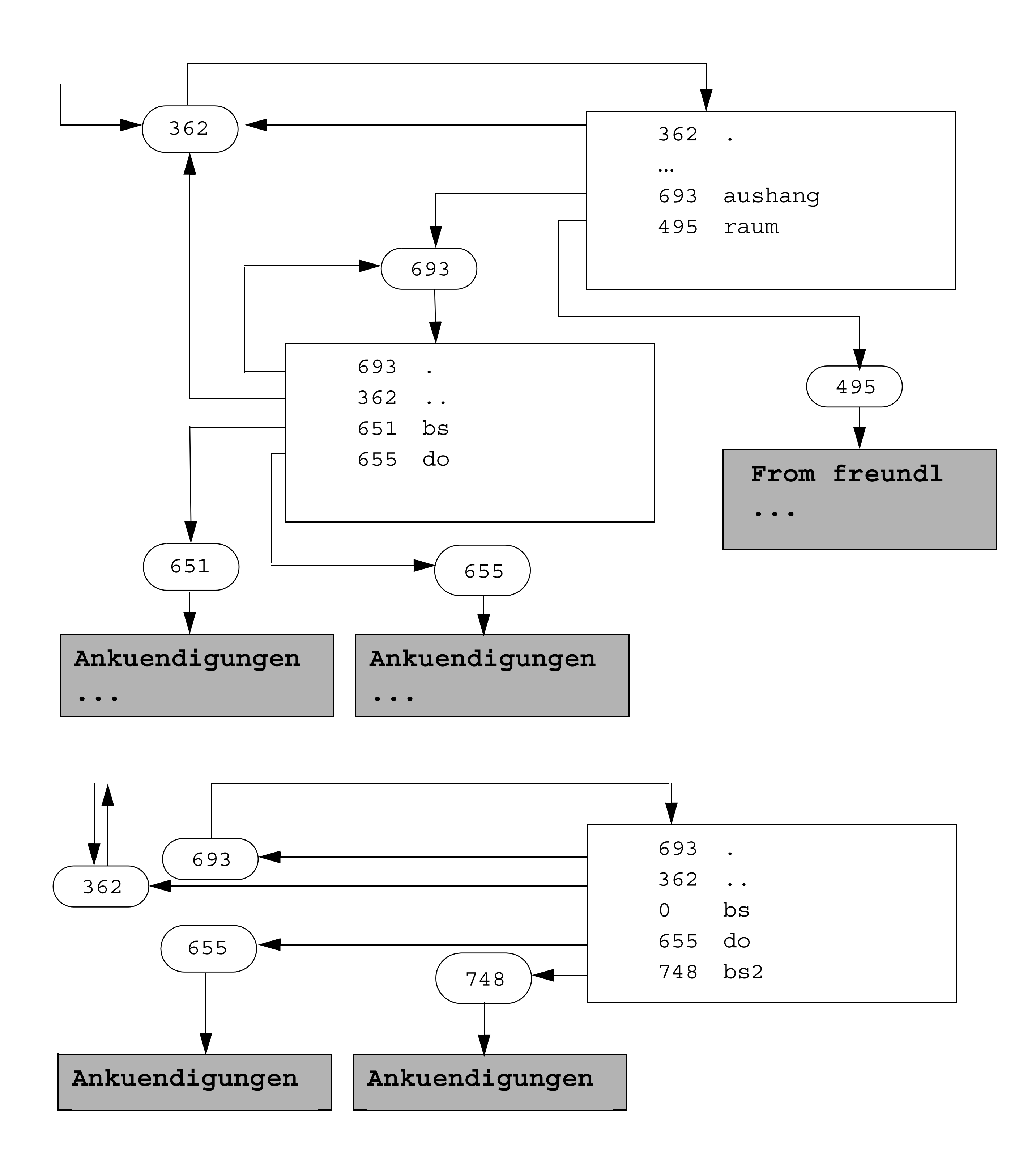
Das Bild unten zeigt das Verzeichnis aushang und sein Vorgängerverzeichnis mit den i-Knoten im Detail.

Für jede physische Datei (Normaldatei, Verzeichnis, Gerät) gibt es genau einen i-Knoten. Er enthält die Dateirechte, den Dateibesitzer, die Adressen der Datenblöcke, usw.

Am Katalog aushang läßt sich die Semantik von cp, mv und ln nochmals erläutern. Zunächst wird die Datei bs in bs2 kopiert.

```
$cd aushang
$cp bs bs2
$ls -i
651 bs
748 bs2
655 do
$
```

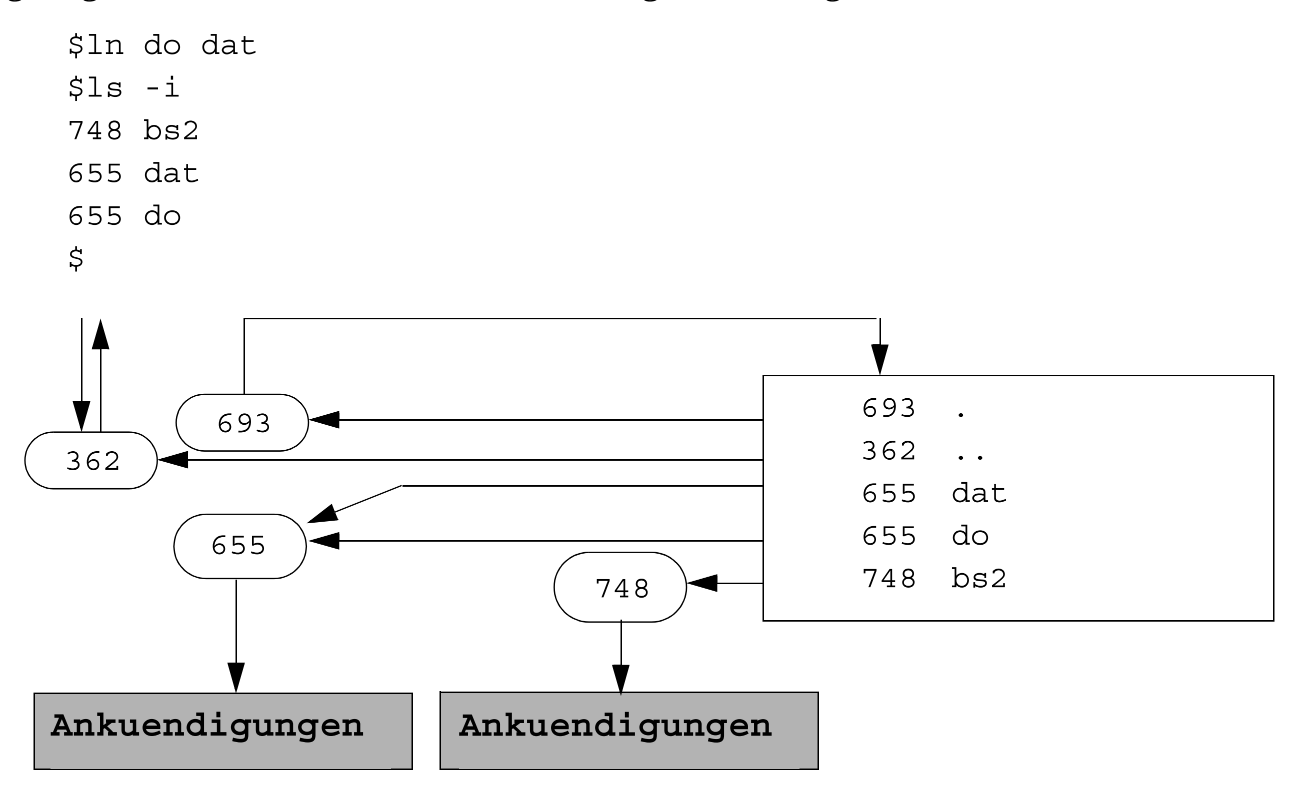
Nun wird die Datei bs gelöscht. Die i-Nummer wird dadurch Null, wie man im Bild unten erkennen kann..


362
362 .
…
693 aushang
495 raum
693
495
693 .
362 ..
651 bs
655 do
From freundl
...
651
655
Ankuendigungen
...
Ankuendigungen
...
693
362
693 .
362 ..
0 bs
655 do
748 bs2
655
748
Ankuendigungen
Ankuendigungen

```
$rm bs
$ls -ai
693 .
362 ..
748 bs2
655 do
$
```

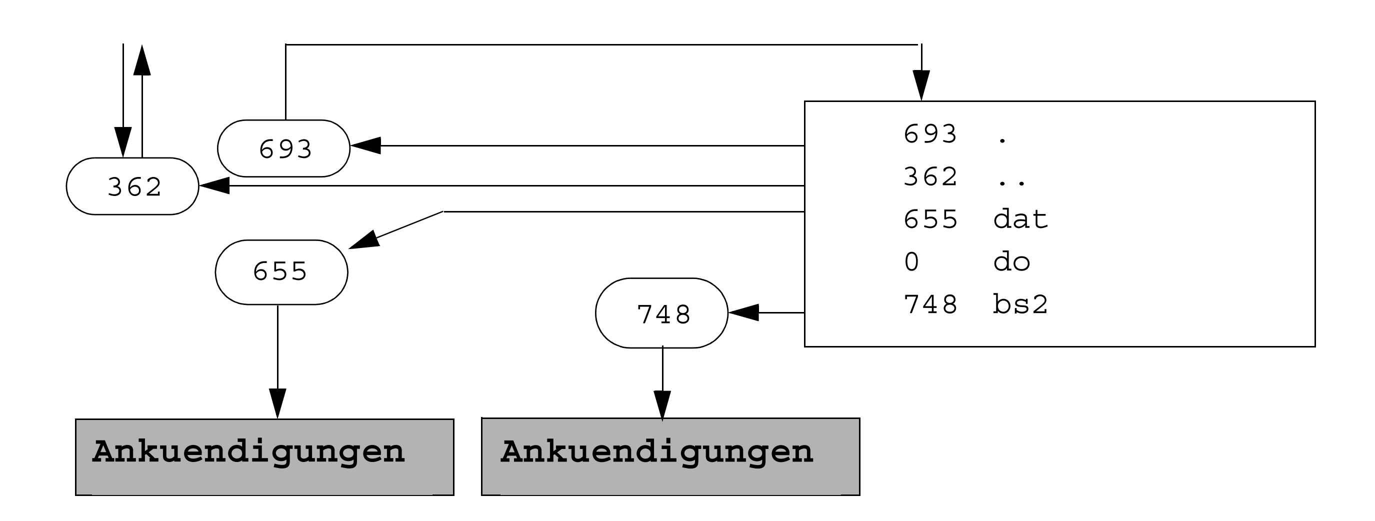
Als nächstes wird jetzt der Name `dat` als zusätzlicher Verweis (Link) auf die Datei `do` eingetragen und überschreibt dabei im Katalog den Eintrag mit der i-Nummer 0.

```
$ln do dat
$ls -i
748 bs2
655 dat
655 do
$
```



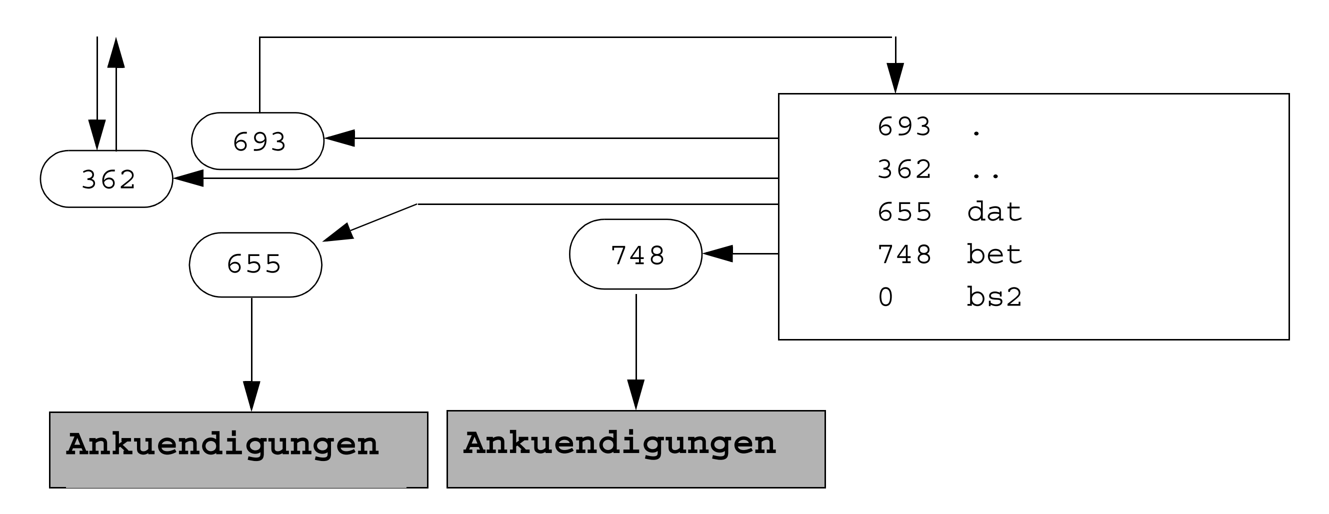
Jetzt löschen wir die Datei `do`. Der Verweis von `dat` auf diese Datei bleibt dabei bestehen, wie man dem Bild unten entnehmen kann. Zuletzt wird dann mittels des `mv` Kommandos die Datei `bs2` in `bet` umbenannt.

```
$rm do
```



```
$ls -i
748 bs2
655 dat
$mv bs2 bet
$ls -i
748 bet
655 dat
$
```

Am letzten Bild kann man sich auch klarmachen, daß `mv bs2 bet` durch die beiden Operationen `ln bs2 bet` und `rm bs2` realisiert wird.



Alle Zeiger auf eine Datei sind gleichberechtigt und nicht unterscheidbar. Die i-Knoten führen einen Zähler für die Anzahl der Zeiger, die auf sie zeigen. Eine Normaldatei wird physisch gelöscht, wenn der Zähler null ist, ein Verzeichnis (wegen *Punkt*), wenn er eins ist.

Hard links versus soft links

Die i-Nummer einer Datei ist nur innerhalb eines *Dateisystems* (volume) eindeutig. Will man einen Verweis auf eine Datei in einem anderen Dateisystem anlegen, muß man einen sog. *symbolischen Verweis* verwenden, den man mit `ln` und der Option `s` (**s**ymbolic) und der selben Syntax und Semantik wie oben erzeugt.

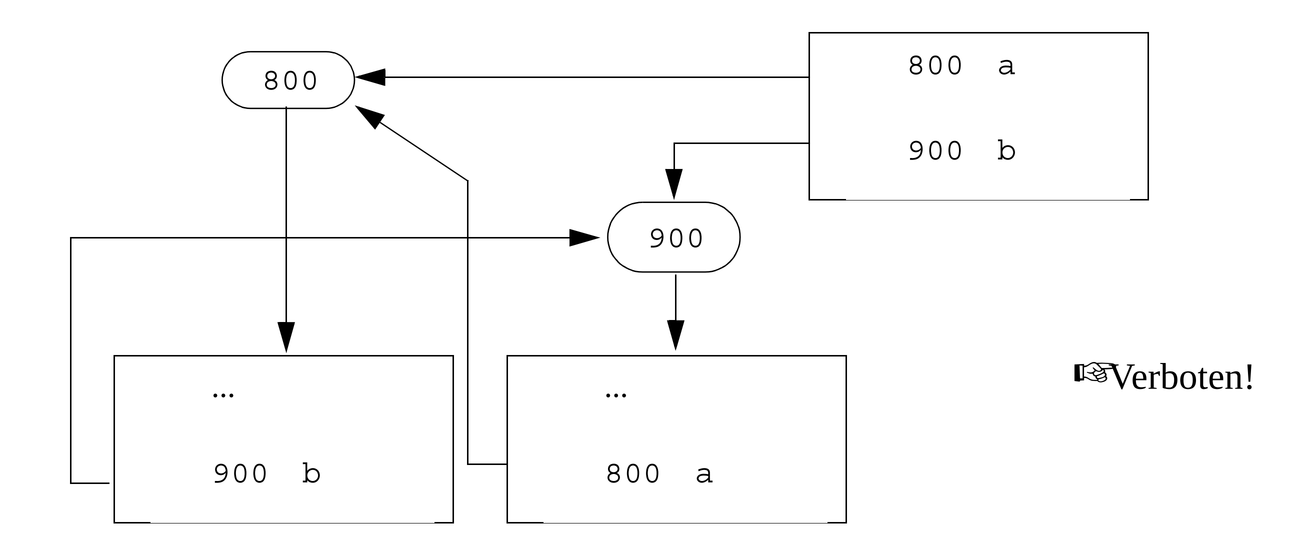
Der Verweis ist dann wie eine normale Datei aufgebaut, die jedoch als Inhalt den Pfadnamen der Datei enthält, auf die der Verweis zeigt. Als Beispiel erzeugen wir `DatenOrg` und `DO` als symbolische Links und `dathard` als „harten“ Verweis auf `dat`.

```
ln -s dat DatenOrg
ln -s /usr/fix/kurs/dat DO
ln dat dathard
ls -l
total 24
lrwxrwxrwx 1 fix mitarb      17 Apr 1 12:31 DO -> /usr/fix/kurs/dat
```

```
lrwxrwxrwx 1 fix mitarb        3 Apr 1 11:51 DatenOrg -> dat
-rw-r--r-- 1 fix mitarb      117 Apr 1 14:30 bet
-rw-r--r-- 2 fix mitarb      117 Apr 1 14:30 dat
-rw-r--r-- 2 fix mitarb      117 Apr 1 14:30 dathard
$
```

Beim Auflisten des Kataloginhalts mit der `l` (**l**ong) Option sehen wir bei den ersten beiden Einträgen vorne das Zeichen `l` als Indikator, daß es sich um einen symbolischen Verweis handelt. Andererseits berücksichtigt die Zählung bei `dathard` und `dat` nur den harten Verweis. In der Tat gibt es bei symbolischen Verweisen keine Garantie, daß die Datei, auf die verwiesen wird, überhaupt existiert. Andererseits kann man symbolische Verweise auf Verzeichnisse zeigen lassen. Zuletzt ist die Byte-Zählung in der 5. Spalte ein Indiz für die Tatsache, daß bei dem 1. Verweis nur der relative Pfadname `dat` abgespeichert wurde, beim 2. Verweis aber der volle Pfadname.

Auf ein Verzeichnis darf `ln` (mit harten Links) nicht angewendet werden. Damit gibt es außer über *Punkt* und *PunktPunkt* keine weiteren Verweise auf einen Katalog. Das Bild zeigt ein hypothetisches Beispiel dafür, welche Folgen sonst eintreten könnten.



☞Verboten!

Vesuchen Sie jetzt `rm -r a` (rekursives Löschen). Damit wäre man böse gelinkt!

Zusammenfassung

❖ Wir haben das Prinzip der i-Knoten kennengelernt und mit `ln` mehrere Verweise (Links) auf eine Datei eingetragen.

Frage 8

```
$pwd
/usr/fix
$mkdir umlauf; cd umlauf
$ln /usr/fbinfo/ausflug umlauf
$
```

In obigem Dialog erzeugt `ln` einen Eintrag

❏ `ausflug` in `/usr/fix`,

❏ `ausflug` in `/usr/fix/umlauf`,

❏ `umlauf` in `/usr/fix/umlauf`,

❏ eine Fehlermeldung.

Frage 9

Ein Verzeichnis `/usr/tricky` enthalte die Einträge für *Punkt* und *PunktPunkt* sowie für 10 andere Verzeichnisse und 3 Normaldateien. Wieviele Verweise zeigen auf den i-Knoten von `/usr/tricky`?

Frage 10

Beim Aufruf des Editors `ed` mit `ed umlauf` arbeitet `ed` mit `/tmp/eumlauf`. Welches Kommando führt der Editor dazu sinngemäß aus?

❏ `cp umlauf /tmp/eumlauf`

❏ `ln umlauf/`

❏ `tmp/eumlauf`

❏ `mv umlauf /tmp/eumlauf`

LEKTION 6: Schutzmechanismen

6.1 Recht(e) haben

❖ In diesem Abschnitt lernen wir das Lese-, Schreib- und Ausführungsrecht für Normaldateien und Verzeichnisse kennen. Wir verwenden `ls` mit der Option `-l` zum Zeigen der Rechte.

Bisher sind wir davon ausgegangen, daß Dateien, die wir lesen, editieren oder ausführen wollten, uneingeschränkt zugänglich waren. Diesen Zustand der paradiesischen Unschuld müssen wir nun verlassen.

Für die Widrigkeiten des Alltags bietet UNIX Schutzmechanismen in drei Ebenen an:

- Rechte der einzelnen Benutzer,
- Rechte des Systemverwalters,
- Rechte des Systemkerns.

Beispiele für den auf den verschiedenen Ebenen möglichen Schutz sind:

- Lesen einer Datei nur mit Erlaubnis des Besitzers,

- Einrichten eines neuen Teilnehmers nur durch den Systemverwalter,
- Verbot des direkten Schreibens in Verzeichnisse für alle.

Die Berechtigten und ihre Rechte werden jeweils in drei Klassen aufgeteilt.

Als Berechtigte können benannt werden:

- Der *Besitzer* (mit `u` für user).
- Die *Gruppe* (mit `g` für group).
- Der *Rest der Welt* (mit `o` für others).

Folgende Rechte können vergeben werden:

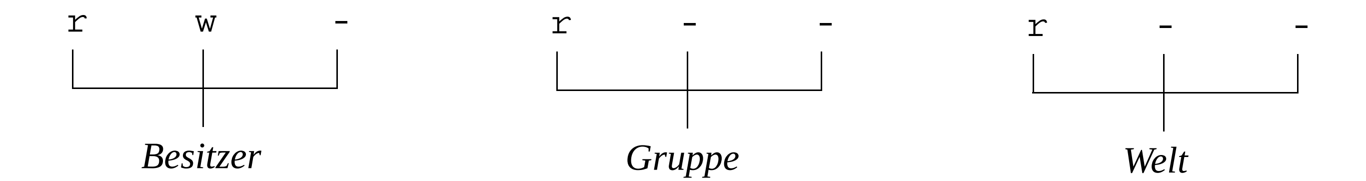
- *Lesen* (mit `r` für read),
- *Schreiben* (mit `w` für write),
- *Ausführen* (mit `x` für execute).

Besitzer (*user*) einer Datei ist, wer sie angelegt hat, sofern nicht das Besitzrecht nachträglich geändert wurde. Eine Gruppe (*group*) bilden alle die Benutzer, für die derselbe Gruppenname vereinbart wurde. Die Gruppenrechte an einer Datei stehen allen Benutzern offen, die Mitglied in der Gruppe des Besitzers der Datei sind. Alle anderen Benutzer (*others*) bilden die dritte Klasse.

Typische Anwendungen für das Sperren des einen oder anderen Rechts sind:

- Die private Korrespondenz (`r` nur für den Besitzer),
- Ein „elektronischer“ Aushang (`r` für alle, `w` nur für den Besitzer),
- Ein Lizenzprogramm (nur `x`).

Die gebräuchliche, von jedem Teilnehmer aber individuell änderbare, Vorbesetzung für neu angelegte Dateien (jeweils `r w x` Angabe) ist



wobei das Minuszeichen für gesperrt steht. Es wird also angenommen, sofern nicht z. B. ein Compiler die Datei für seinen Zielcode anlegt, daß die Datei nicht ausführbar ist und nur der Besitzer sie ändern darf. Die Schutzmechanismen sind einheitlich für alle Dateien, also auch für Geräte und Verzeichnisse.

Für Verzeichnisse muß die Interpretation der Rechte[1] naturgemäß anders sein als für Normaldateien:

1. und die Vorbelegung, nämlich `rwxr-xr-x`,

- Lesen (r) Inhalt des Katalogs einsehen, z. B. mit `ls`.
- Schreiben (w) Inhalt des Katalogs ändern, z. B. Dateien anlegen und löschen.
- Ausführen (x) Auf Einträge zugreifen und z. B. das Verzeichnis mit `cd` zum aktuellen Katalog machen, aber ohne das Leserecht (r) ist kein Auflisten mit `ls` möglich.

Ist der Rechner stark belastet, kann der super-user z. B. `/usr/games` auf `r - -` setzen: „what you see is not what you get!“

Wichtig ist dabei:

- Ist eine Datei schreibgeschützt, kann immer noch ihr Eintrag im Verzeichnis gelöscht werden — womit auch die Datei unwiderruflich weg ist. Um dies zu verhindern, muß auch das Verzeichnis selbst schreibgeschützt sein.

Der Schlüssel zum Betrachten der Rechte ist wieder das Kommando `ls`, diesmal mit der Option `l` (long), wobei wir die Anzeige hier im Text geringfügig gekürzt haben.

```
$pwd
/usr/fix
$ls -l
drwxr-xr-x 2 fix     128 Feb  6 aushang
-rw-r--r-- 1 fix      35 Dec 30 nachb3
-rw-r--r-- 1 fix     176 Feb  6 raum
-rw-r--r-- 9 dekan   989 Feb 12 StO97
$
```

Die erste Zeile der Ausgabe von `ls -l` zeigt folgende Informationen an:

- `d` Dateityp, hier Katalog,
- `rwx` Rechte des Besitzers (alle Rechte),
- `r-x` Rechte der Gruppe (kein Schreibrecht),
- `r-x` Rechte der anderen (kein Schreibrecht),
- `2` Anzahl der Verweise auf die Datei,
- `fix` Besitzer (üblicherweise auch die Gruppe),
- `128` Größe der Datei in Bytes,
- `Feb 6` Datum der letzten Änderung (üblicherweise auch die Uhrzeit),
- `aushang` Name der Datei.

Als *Dateityp* können die Angaben `d` für einen Katalog, Minuszeichen für eine Normaldatei, `b` für ein block- und `c` für ein zeichenorientiertes Gerät, `l` für einen symbolischen Verweis, `p` für eine benannte Pipe, `s` für ein *Socket* und `m` für ein *Speichersegment* (memory) auftreten.

Besitzer der „Studienordnung“ `StO97`, die sich Professor Fix früher über `ln` geholt hat, ist ein Benutzer `dekan` (mit der Wahl zum Dekan erhält man das Paßwort vom Vorgänger im Amt). Er hat die Datei `StO97` schreibgesperrt für alle außer ihn selbst.

Mit `ls -lg` erhält man die Gruppenzugehörigkeit der Datei, hier die „Studienkommission“ `stkom`.

```
$ls -lg StO97
-rw-r--r-- 9 stkom   989 Feb 12 StO97
$
```

Mit `ls -ld` erhalten wir die lange Ausgabe eines Katalogs selbst, also nicht die seiner Einträge.

```
$ls -ld /usr/fbinfo
drwxr-xr-x 4 dekan   128 Aug 10 /usr/fbinfo
$
```

Das Verzeichnis `/usr/fbinfo` ist, wie der Eintrag `StO97` selbst, schreibgeschützt, aber man kann es (`x` Recht) ohne weiteres zum aktuellen Katalog machen.

```
$cd /usr/fbinfo
$rm StO97
rm: StO97: 644 mode y
rm: StO97 not removed
$
```

Beim Entfernen erwartet `rm` eine Antwort, weil `StO97` schreibgeschützt ist. Fix antwortet mit `y` (für yes), aber der Eintrag läßt sich nicht entfernen. Dann eben den Inhalt löschen!

```
$echo Weg mit der Ordnung >StO97
StO97: cannot create
$
```

Auch das geht nicht.

Im Zusammenhang mit `rm StO97` erhielt Professor Fix den Hinweis, daß die Schutzrechte den Wert `644` haben. Dies ist ein Relikt aus der Zeit, als jeder Programmierer seine Bits noch selbst verwaltete. Jedes Recht wird im i-Knoten einer Datei durch ein Bit (bestehendes Recht gleich 1) dargestellt, d. h. `rw-r--r--` entspricht `110100100` (dual) bzw. `644` (oktal).

Kann Professor Fix ein Verzeichnis aus `/usr` entfernen oder einen neuen Katalog, wie etwa `fbinfo` für die Fachbereichs-Informationen, dort anlegen?

```
$ls -ld /usr
drwxr-xr-x 25 root   432 Sep  1 /usr
$mkdir /usr/tester
mkdir: cannot access /usr/.
$ls -dl /usr/fix
drwxr-xr-x 3 fix     144 Oct 11 /usr/fix
$
```

Nein, auch das darf er nicht. Professor Fix ist zwar, wie zu erwarten, Besitzer seines Heimatverzeichnisses `/usr/fix`, Besitzer von `/usr` ist aber ein Benutzer namens `root` (Wurzel). Das ist der traditionelle login-name des super-users.

UNIX wäre nicht UNIX, wenn man nicht für den Systemschutz wieder eine einfache, einheitliche Lösung gefunden hätte.

- Systemdateien werden mit dem gleichen Mechanismus geschützt wie Benutzerdateien. Besitzer der nicht öffentlichen Dateien ist der super-user.

Wie man seine eigenen (bescheidenen) Rechte verwaltet, zeigen die nächsten Abschnitte. Wie man an den beinahe allmächtigen Rechten des super-users teilhaben kann, bringt die nächste Lektion.

Zusammenfassung

❖ Wir lernten das Lese- (`r`), das Schreib- (`w`) und das Ausführungsrecht (`x`) für Normaldateien und Verzeichnisse kennen sowie die Unterteilung in Besitzer (`u`), Gruppe (`g`) und Welt (`o`). Mit `ls -l` sehen wir die Dateirechte.

Frage 1

Um eine Datei wirklich zu schützen, muß man auch das Verzeichnis mit dem Eintrag der Datei schreibschützen. Muß man auch dessen Vorgängerverzeichnis usw. schützen?

❏ Ja, das muß der Besitzer veranlassen.

❏ Ja, das veranlaßt das System automatisch.

❏ Nein, das ist nicht nötig.

❏ Nein, das ist nicht möglich.

Frage 2

Das Verbot, Kataloge mit dem Editor zu ändern, ist geregelt durch

❏ den Besitzer des Katalogs (`w`-Recht),

❏ den super-user (ihm vorbehalten),

❏ den Betriebssystemkern.

Frage 3

Die übliche Vorbesetzung der Rechte für Normaldateien war `644` (`rw-r--r--`). Wie lautet die übliche Vorbesetzung für Verzeichnisse als Oktalzahl?

6.2 Ausführungsbestimmungen

❖ In diesem Abschnitt ändern wir Rechte mit `chmod`, um Dateien ausführbar zu machen oder zu schützen. Wir lernen ferner das Konzept des Shell-Skripts kennen.

Ein dickes Benutzerhandbuch muß nicht immer ein gutes Zeichen sein — ein klares Konzept braucht wenig Worte. Nach dieser Maxime haben wir bei den Kommandos, z. B. `cat`, `ed`, `cp`, `mv` und `ln`, nichts über Verbote und Besitzveränderungen gesagt.

Wir behaupten einfach, UNIX mache das, was zu Recht vermutet wird:

- bei `mv` und `ln` übernimmt man die Rechte des alten Eintrags,
- bei `cp` erhält die Kopie die eingestellten Standardrechte für neuangelegte Dateien[1],
- was man lesen darf, kann man auch kopieren, und man wird Besitzer der Kopie,
- mit `ln` und `mv` ändert man nicht den Besitzer usw.

Die wesentliche Regel ist dabei, daß nur der Besitzer — und wie immer der super-user — Rechte ändern und den Besitz übertragen darf. Ersteres erledigt das Kommando `chmod` (**ch**ange **m**ode), letzteres die Kommandos `chown` (**ch**ange **own**er) bzw. `chgrp` (**ch**ange **gr**ou**p**). Der Gebrauch von **chmod** ist umständlich.

Die Syntax lautet

`chmod` *rechte datei ...*

wobei *rechte* die Form [`ugoa`] [`+-=`] [`rwxstugo`] hat.

Die Buchstaben `u`, `g`, `o`, und `a` beziehen sich auf Benutzerklassen, die Zeichen `+`, `-` und `=` auf Berechtigungsänderungen und `r`, `w`, `x`, `s`, `t`, `u`, `g` und `o` auf die Rechte selbst.

Allerdings können die Rechte auch direkt oktal angegeben werden, etwa durch `chmod 666 kein-geheimnis` oder `chmod 111 probier-mich`.

Die Rechte `s` und `t` gehören zu drei weiteren Bits, die vor den neun bereits bekannten für `rwx` von Besitzer, Gruppe und Welt im i-Knoten einer Datei stehen. Sie heißen (von links nach rechts):

- `SUID` (set-user-id),
- `SGID` (set-group-id)
- *saved text* (sticky bit) zum Halten von Textsegmenten im Haupt- bzw. Swapspeicher.

Angezeigt werden sie durch `s` statt `x` für Besitzer und Gruppe bzw. `t` statt `x` für die übrige Welt, wobei noch anzumerken ist, daß `s` und `t` bei Verzeichnissen eine etwas andere Bedeutung haben. Hier einige Dateien, bei denen diese speziellen Bits gesetzt sind.

```
$ls -l /bin/sh /bin/mkdir bin/passwd
```

1. je nach der mit `umask` eingestellten Maske, meist oktal 022, die zu löschende Rechte angibt (vgl. Lektion 10)

Benutzerklasse

u	Besitzer (user)
g	Gruppe (group)
o	Welt (others)
a	Alle (all)

Berechtigung

+	erteilen
-	entziehen
=	direkt setzen

Rechte

r	Lesen
w	Schreiben
x	Ausführen
s	Besitzer- und Gruppennummer annehmen
t	Textsegment halten
ugo	Rechte übernehmen von bisheriger Besetzung

```
-r-x--x--t 1 bin      3502 /bin/sh
-r-s--x--x 1 root     9814 /bin/mkdir
-rws--s--x 1 root    18402 /bin/passwd
$
```

In diesem Beispiel haben die Shell als ständig gebrauchtes Programm das `t` Bit[1] und die Kommandos `mkdir` und `passwd` das `s` Bit gesetzt (siehe Lektion 7).

Professor Fix erstellt seine Klausur.

```
$cat >klausur
Klausur zur Vorlesung
UNIX
Beantworten Sie alle 247 Fragen!
Hilfsmittel sind nicht zugelassen.
CTRL-d
$chmod go-rw klausur
$ls -l klausur
-rw------- 1 fix    104 Feb 16 klausur
$
```

Der Gruppe (`g`) und den anderen (`o`) wird das Lese- und Schreibrecht entzogen — for his eyes only!

Als nächstes erzeugt Professor Fix zwei „selbstgestrickte“ Kommandos: `mkfile` (Datei anlegen) und `l2` (Verzeichnis zweispaltig auflisten).

1. Das sticky bit wird nicht vom POSIX-Standard unterstützt und ist auf modernen Rechnern, die Paging für die Hauptspeicherverwaltung verwenden, nicht mehr üblich.

```
$echo 'echo -n >$1' >mkfile
$echo 'ls $* | pr -2t' >l2
$cat mkfile l2
echo -n >$1
ls $* | pr -2t
$ls -l mkfile l2
-rw-r--r-- 1 fix 11 Feb 16 mkfile
-rw-r--r-- 1 fix 14 Feb 16 l2
$
```

Das Symbol `$1` steht dabei für das erste Argument, `$*` für alle Argumente, die im Aufruf von `mkfile` bzw. von `l2` auftreten.

```
$mkfile test
mkfile: cannot execute
$
```

Der Aufruf von `mkfile` ohne `x` Recht mißlingt.

```
$chmod +x mkfile l2
$mkfile Pro87; l2
PrO87                mkfile
StO97                nachb3
aushang              raum
$
```

So funktioniert es jetzt[1]. Statt `chmod +x` wären z. B. auch `chmod a+x`, `chmod ugo+x`, `chmod u+x`, `g+x`, `o+x` und alle Varianten mit `=rx` möglich gewesen.

Ausführbare Dateien, die Kommandozeilen enthalten, nennt man *Shell-Skripte*. Erweitert um Parametermechanismen, Variablen und einige Ablaufkontrollstrukturen stellen sie ein mächtiges Programmierwerkzeug dar. Mehr dazu und eine bessere Version von `mkfile` später.

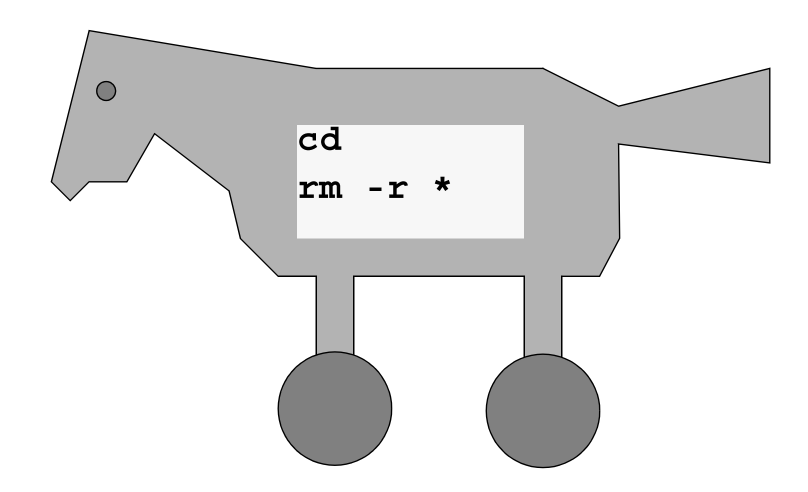
Generell sollte man nicht jede Datei ausführbar machen. Ein harmloses Beispiel wäre eine unsortierte Liste einiger Professor Fix bekannter Kommandos in der Datei `komm`. Ihre ungewollte Ausführung erzeugt zumindest ein Aha-Erlebnis und verursacht größere Aufräumarbeiten.

```
$cat komm
## Dies ist eine Liste von Kommandos ##
mkdir ls write pwd cd lpr
nice echo cat diff tr date ed
$chmod +x komm
$komm
cat diff tr date ed
$
```

1. Vorausgesetzt der Pfad ist richtig eingestellt, ggf. Aufruf mit vollem Pfadnamen.

Große Vorsicht ist auch beim Ausführen unbekannter Dateien geboten. Besonders einen geschenkten Gaul muß man sich genauer ansehen.

Entlarvt man die Datei dann z. B. als ein Shell-Skript wie das im nächsten Bild, handelt es sich dabei um ein hochgefährliches *Trojanisches Pferd*.



Auch wenn UNIX-Systeme gerne offen geführt werden, sollten folgende Regeln stets beachtet werden:

- Das aktuelle Verzeichnis *Punkt* sollte nicht im Suchpfad für Kommandos liegen.
- Verzeichnisse und Normaldateien sind schreibgeschützt für alle außer dem Besitzer.
- Von nicht schreibgeschützten Dateien gibt es eine schreibgeschützte Kopie in einem schreibgeschützten Verzeichnis.
- Vertraulicher Inhalt ist verschlüsselt zu speichern.

Den letzten Rat sollte auch Professor Fix für die Datei `klausur` beherzigen. Im nächsten Abschnitt erläutern wir das entsprechende Kommando.

Zusammenfassung

❖ Wir haben Dateirechte mit `chmod` geändert und einfache Beispiele für die Erstellung und den Gebrauch eines Shell-Skripts kennengelernt. Möglichkeiten des Dateischutzes wurden angesprochen.

Frage 4

Auf älteren UNIX-Systemen hätte sich Professor Fix mit der Zeile `cd; chmod -x .` beim Systemverwalter sehr unbeliebt gemacht. Warum?

❑ Der Systemverwalter konnte die Kataloge von Professor Fix nicht mehr lesen.

❑ Der Systemverwalter konnte die Kataloge von Professor Fix nicht mehr betreten.

❑ Professor Fix konnte sich nicht mehr anmelden (einloggen).

Frage 5

Das einfache Shell-Skript `mkfile` hatte den Inhalt `echo -n >$1`. Wie erreicht man, daß zusätzlich bei dem Aufruf `mkfile` *prog* die Rechte von *prog* auf `rwxr-xr-x` gesetzt werden?

❏
```
echo -n >$1
chmod 755 mkfile
```

❏
```
echo -n >$1
chmod 755 $1
```

❏
```
chmod 755 mkfile
echo -n >$1
```

Frage 6

Die Datei `komm` ist irrtümlicherweise ausführbar. Ihr Inhalt lautet:

```
komm ist eine
Datei, die ...
```

❏ Der Aufruf von `komm` ist harmlos (Fehlermeldung),

❏ führt zu einer endlosen Rekursion,

❏ ist harmlos (`komm` wird einmal durchlaufen),

❏ wird nicht ausgeführt, da `komm` ein Text und kein Programm ist.

6.3 Ein offenes Geheimnis

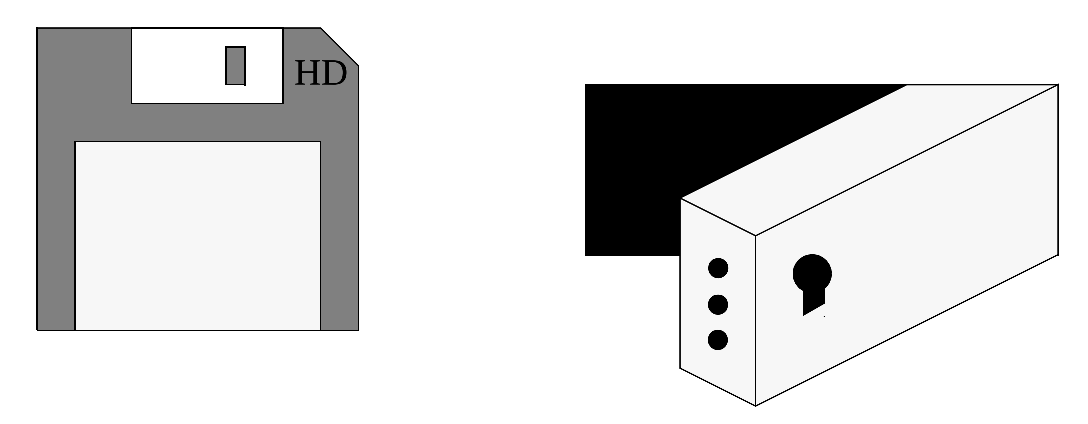
❖ In diesem Abschnitt lernen wir die Verschlüsselung von Daten mit dem Kommando `crypt` kennen und lesen in der Paßwortdatei.

Jedes (Betriebs-) System hat seine allgemein bekannten, weniger bekannten und noch nicht entdeckten Sicherheitslücken. UNIX gehört zu den eher sicheren Systemen, hat aber wegen der uneingeschränkten Macht des super-users eine eingebaute Schwachstelle.

Um zuverlässig den Verlust von Daten zu vermeiden, gibt es nach allgemeiner Einschätzung nur eine (teure) Methode der Sicherung:

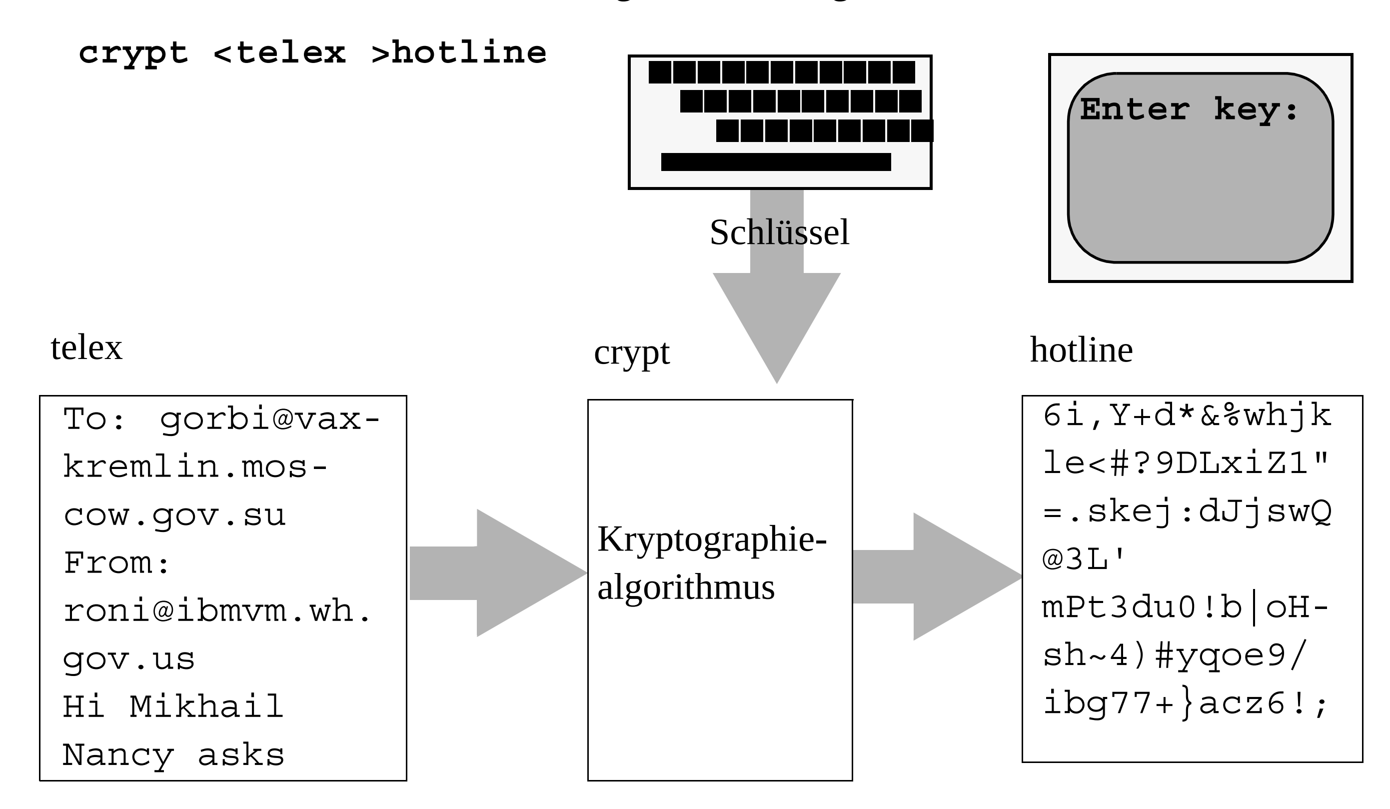
Gegen die unbeabsichtigte Preisgabe von Daten gibt es aber erfreulicherweise eine schnellere und billigere Methode: das Verschlüsseln.

Das Kommando crypt <*klartext* >*codetext* leistet dies. Sein Schema wird im nächsten Bild am Beispiel crypt <telex >hotline gezeigt. Es leistet sowohl die Verschlüsselung als auch die Entschlüsselung einer Datei. `crypt` ist verträglich mit der `x` Option bei



den Editoren vi und ed. Nach Europa ausgelieferte Versionen von UNIX enthalten `crypt` leider oft nicht.

Der in `crypt` implementierte Algorithmus simuliert eine Verschlüsselungsmaschine nach dem Vorbild der deutschen Enigma aus dem 2. Weltkrieg (256 Elemente, 1 Rotor). Die Methode ist nicht so sicher wie man lange dachte und gilt heute als nicht ausreichend.



Die Verschlüsselungsdebatte

Verschlüsselung ist in UNIX ein „heißes Thema“, dem in den letzten 20 Jahren viel Aufmerksamkeit geschenkt wurde. Die Gründe sind:

- die Vernetzung der Rechenanlagen, spez. die Möglichkeit sich an einem fremden Rechner, z. B. über `telnet`, anzumelden,
- die Verbreitung von UNIX-Rechnern in kommerziellen und wissenschaftlichen Bereichen mit notwendigerweise sensiblen Daten,

- die Möglichkeit über Netze Geschäfte abzuwickeln und damit die Notwendigkeit, *digitale Unterschriften* nachprüfbar zu machen,
- das Aufkommen neuer Verschlüsselungsmethoden, spez. der sog. *public key Algorithmen*, die digitale Unterschriften erlauben und das Problem des Verschickens der geheimen Schlüssel vermeiden,
- der rasante Anstieg der Rechenleistung bei PC's und Workstations gegenüber den frühen Sechzigern, wodurch ein Entschlüsselungsversuch, der damals drei Jahre gebraucht hätte und damit uninteressant war, heute innerhalb eines Tages vermutlich zum Erfolg führen würde,
- zuletzt die Eigenheit, daß kryptographische Angriffe sich gut parallelisieren lassen und daß ein geübter Bastler mit Microprozessoren, die man „beim Trödler an der Ecke" für unter 20 DM das Stück bekommt, oder mit einem großen Netz preiswerter PC's, etwa dem einer Uni, relativ leicht einen kryptographischen Superrechner bauen bzw. betreiben kann.

Neben diesen eher technischen Gründen gibt es aus den USA kommend drei weitere, politisch-juristische Faktoren:

- kryptographische Verfahren gelten als Rüstungsgüter; ihr Export unterliegt einem Verbot, das auch nach Ende des Kalten Krieges noch gilt;
- fast alle der neueren Verfahren sind patentrechtlich geschützt, spez. durch das *Rivest-Shamir-Adleman* (RSA) Patent, und damit für die freie Verbreitung im Internet nicht geeignet;
- die amerikanische Regierung beschloß 1994 auf Drängen des Geheimdienstes (NSA, National Security Agency) eine Verschlüsselung, die ihr eine Tür zur Entschlüsselung, kontrolliert durch Gerichte, offenhält. Der angestrebte Electronic Escrow Standard (EES), der mittels des sog. *Clipper-Chips* auch für Telephone, Fax und alle sonstige digitale Kommunikation gilt, wird von weiten Teilen der auf Einhaltung der Privatsphäre pochenden Informatiker abgelehnt.

Als Alternative, die nach jüngsten Gerichtsbeschlüssen von den obigen Einschränkungen nicht berührt ist, bietet sich das von *Phil Zimmermann* entwickelte Verfahren *Pretty Good Privacy* (PGP) an, auf das weiter unten eingegangen wird. Die hier zu `crypt` gemachten Anmerkungen gelten sinngemäß auch für PGP.

Professor Fix schaut sich die ersten vier Zeilen des Klartextes an.

```
$head -4 klausur
Klausur zur Vorlesung
UNIX
Beantworten Sie alle 247 Fragen!
Hilfsmittel sind nicht zugelassen.
```

```
$
```

Dann ruft er `crypt` auf.

```
$crypt <klausur >cklausur
Enter key:
$
```

Während der Eingabe des Schlüssels ist das Echo ausgeschaltet, d. h. die eingegebenen Zeichen werden nicht auf dem Bildschirm angezeigt. Wichtig ist die Kontrolle, daß man sich nicht vertippt hat!

```
$crypt <cklausur | diff - klausur
Enter key:
$
```

Das Kommando `diff` (mit dem Minuszeichen als erstes Argument wird als Eingabe die Standardeingabe bestimmt, in diesem Fall der Eingabestrom der Pipe) stellt keinen Unterschied fest, also kann klausur weg.

```
$rm klausur
$
```

Die Ausgabe der verschlüsselten Dateien auf dem Bildschirm empfiehlt sich nicht, da hierbei unter Umständen Zeichen oder Zeichenkombinationen auftreten, auf die ein Terminal schlimmstenfalls mit Sperren der Tastatur reagieren kann, bestenfalls wird es laut, weil ASCII 007 (`BEL`) öfters ertönt.

Eine ziemlich gute Verschlüsselungsmethode

Wollen Sie ein Dokument vor dem Lesen durch andere schützen, können Sie es mit einem — nur Ihnen bekannten — Paßwort verschlüsseln. Möchten Sie das verschlüsselte Dokument aber einer weiteren Person zugänglich machen, können Sie nicht nur das verschlüsselte Dokument verschicken, die Person muß auch Ihr Paßwort kennen. Hätten Sie aber einen sicheren Übertragungsweg für das Paßwort, könnten Sie auch gleich das (unverschlüsselte) Dokument übertragen.

Eine Lösung für dieses Problem bietet das von *Phil Zimmermann* entwickelte Verfahren *Pretty Good Privacy* (PGP), das mehrstufig und mit sog. allgemeinen und privaten Schlüsseln arbeitet. Jeder Benutzer von PGP hat einen privaten Schlüssel (eine längere Binärfolge), der nur ihm selbst bekannt und durch ein Paßwort geschützt ist, und einen dazu „passenden“, öffentlich bekannten Schlüssel. Außerdem hat er einen Schlüsselring, in dem die öffentlichen Schlüssel anderer Personen gespeichert sind.

Soll ein geheimer Text an eine bestimmte Person verschickt werden, so benutzt man dessen allgemeinen Schlüssel zum Verschlüsseln mit PGP. Nur mit seinem zugehörigen privaten Schlüssel (den man normalerweise selbst auch nicht kennt) kann der Empfänger den resultierenden Code-Text wieder entschlüsseln. Die Antwort an einen selbst wird auf die gleiche Weise verschickt: der Gegenüber benutzt unseren allgemeinen Schlüssel und nur wir

selbst können mit unserem privaten Schlüssel die Antwort wieder entschlüsseln. Der Vorteil ist, daß nur Code-Texte und sowieso allgemein bekannte Schlüssel übertragen werden müssen.

Das als public domain Programm verfügbare PGP (URL `http://web.mit.edu/network/pgp-form.html/`) bietet noch viele weitere Funktionen, z. B. um mehrere Adressaten anzusprechen, die Authentizität von Texten sicherzustellen und den Schlüsselring zu verwalten. Durch ein sog. „Netz des Vertrauens“ kann die Sicherheit der allgemeinen Schlüssel erhöht werden, wodurch sich der Gefahr begegnen läßt, daß öffentliche Schlüssel von Lauschern einer Übertragung verfälscht werden, um eine später übertragene und ebenfalls abgefangene Botschaft zu entschlüsseln. Die Details können etwa dem Buch von *Simson Garfinkel* [4] entnommen werden.

Mit Verschlüsselungen arbeitet auch das Betriebssystem selbst, z. B. in der Paßwortdatei. Dabei kommt eine C-Funktion `crypt()` zur Anwendung, die den *Data Encryption Standard* (DES) des National Bureau of Security (NBS) implementiert.

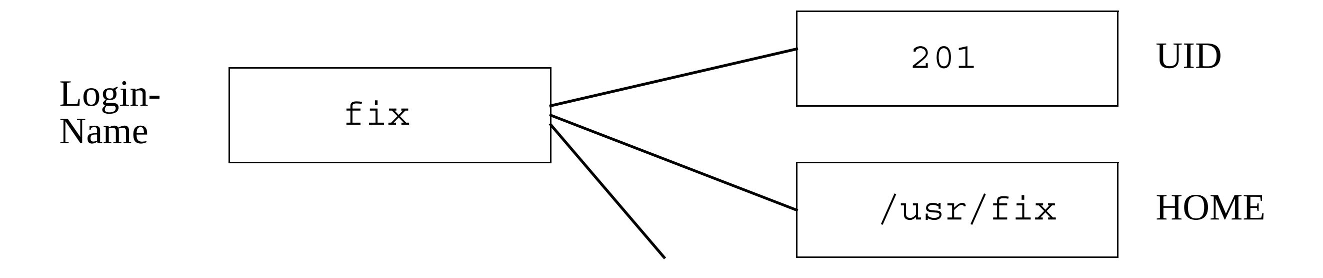
Im Gegensatz zu anderen Betriebssystemen hielt UNIX bis vor wenigen Jahren den Namen der Paßwortdatei nicht geheim und sie war für jedermann lesbar, einschließlich der verschlüsselten Paßwörter, wie unten dargestellt.

```
$ls -l /etc/passwd
-rw-r--r-- 1 root 1324 May 29 /etc/passwd
$cat /etc/passwd | tail -4
fix:R9VtM:201:50::/usr/fix:/bin/sh
neu:j987Y:202:50::/usr/jr:/bin/sh
jr:40iHS:203:50::/usr/jr:/bin/sh
dekan:tG6ce:204:50::/usr/dekan:/bin/sh
$
```

Jede Zeile entspricht einem Teilnehmer. Die Zeilen haben ein festes Format, die verschiedenen Einträge sind durch einen Doppelpunkt voneinander getrennt. Von links nach rechts enthalten sie folgende Angaben:

- login-Name,
- verschlüsseltes Paßwort,
- Teilnehmerkennung (`user id`, `uid`),
- Gruppenkennung (`group id`, `gid`),
- Kommentar (hier jeweils 0 Bytes lang),
- Heimatverzeichnis,
- login-Shell (Standard-Shell).

Die Paßwortdatei `/etc/passwd` stellt den Bezug her zwischen dem *Teilnehmernamen* (login-Name) und der *Teilnehmerkennung*, dem *Heimatverzeichnis*, usw.



Die Paßwortdatei wird von vielen Kommandos konsultiert, z. B. `ls`, `mail`, `write`, `login` und natürlich `passwd`.

Inzwischen werden die Paßwörter häufig getrennt in einer nicht jedermann zugänglichen Datei, z. B. in `/etc/shadow` abgespeichert (siehe Kommentar zur Wahl eines sicheren Paßworts unten).

Weiterhin gilt: Jeder kann sein eigenes Paßwort ändern, nur der super-user kann die anderer Teilnehmer löschen und ein neues einsetzen. Wegen der Verschlüsselung kennt auch er die Paßwörter der Teilnehmer nicht und kann sie deshalb auch nicht rekonstruieren, wenn jemand sein Paßwort vergessen hat.

Warum die Wahl eines guten Paßwortes wichtig ist

Der DES-Verschlüsselungsalgorithmus für die Paßwörter in UNIX gilt als sehr sicher, d. h. es ist eigentlich unmöglich, nur mit Kenntnis des Verfahrens und des verschlüsselten Textes den Klartext wieder herzustellen. Trotzdem können Paßwörter „geknackt“ werden. Warum?

Erstens sind Paßwörter kurz, zweitens gibt es Gewohnheiten für die Wahl von Paßwörtern. So wählen viele Anwender Vor-, Nach- und Ländernamen, geographische oder wissenschaftliche Begriffe, Geburtsjahre, usw. Gegebenenfalls variieren sie diese Namen durch angehängte Ziffern und einen Großbuchstaben. Genau damit aber öffnen sie die Tür für einen „kryptographischen Angriff“.

Public domain Programme, wie z. B. `crack`, verfügen über Bibliotheken von häufig benutzten Paßwörtern und Methoden, diese Bibliothekseinträge wie angedeutet zu variieren. Jedes so erzeugte Paßwort wird verschlüsselt und die Verschlüsselung mit den Einträgen der Paßwortdatei verglichen. Stimmt die erzeugte Verschlüsselung mit der abgespeicherten überein, ist das Paßwort erraten und damit geknackt worden.

Systemverwalter und Sicherheitsbeauftragte von Rechnersystemen können diese Programme nutzen, um ihre Teilnehmer über die Wahl schlechter Paßwörter aufzuklären[1].

Häufig werden sie auch den periodischen Wechsel von Paßwörtern, spätestens alle 6 Monate, überwachen.

Wie kann man nun solchen „brute force“ Methoden vorbeugen? Verwenden Sie ein Paßwort, das nur schwer erraten werden kann. Es sollte aus Klein- und Großbuchstaben, gemischt mit Satzzeichen und Ziffern bestehen, z. B: „Wg:JkseP“ - aber das kann sich doch niemand merken! Doch, denn es ist die Abkürzung des Satzes „Weiterhin gilt: Jeder kann sein eigenes Paßwort ...“ — eine gute Methode, um sich ein kryptisches, aber merkbares Paßwort zu suchen.

Zusammenfassung

❖ Wir lernten die Verschlüsselung von Daten mit dem Kommando `crypt` und den Inhalt sowie die Verwendung der Paßwortdatei `/etc/passwd` kennen.

Frage 7

Welche der folgenden Aussagen im Zusammenhang mit `crypt` ist (sind) richtig?

❏ Das Zeilenendezeichen (newline) wird unverschlüsselt übernommen.

❏ Ist das `crypt` Kommando nicht vorhanden, kann man sich selbst eines schreiben unter Verwendung eines beliebig guten Verfahrens.

❏ Der Schlüssel kommt von der Standardeingabe.

Frage 8

Empfiehlt sich die folgende Methode zur Prüfung der korrekten Eingabe des Schlüssels (der Text der Datei `wichtig` sei bekannter Klartext)?

```
$crypt <wichtig >geheim
Enter key: (Schlüssel eingeben)
$crpyt <geheim >wichtig
Enter key: (Schlüssel nochmals eingeben)
$cat wichtig
```

❏ Ja, sehr zu empfehlen.

❏ Nur für super-user empfehlenswert.

❏ Das ist schon syntaktisch falsch.

❏ Die Methode ist nicht zu empfehlen.

---

1. In einem jüngst durchgeführten Kurs für UNIX-Anfänger konnten wir vor unseren warnenden Worten über die Wahl eines guten Paßwortes von zehn eingetragenen Paßwörtern fünf knacken.

Frage 9

Beim Anmelden „springt das System nicht an“ (Fehlermeldung `No Shell`):

- ❏ Diesen Fehler können sie selbst beheben.
- ❏ Der super-user muß sich mal Ihren Eintrag in der Paßwortdatei vornehmen oder der Kommandointerpreter ist defekt.
- ❏ Momentan sind alle Kommandointerpreter belegt, bitte warten und dann nochmals `login` versuchen.
- ❏ Rufen Sie `take BP` auf.

LEKTION 7: Besitzverhältnisse

7.1 Beziehungen

❖ In diesem Abschnitt lernen wir Teilnehmer und Gruppen und ihre Beziehungen zu Prozessen und Dateien kennen. Wir verwenden die Kommandos `chown`, `chgrp`, `newgrp` und `su`.

In Lektion 6 sahen wir die Einträge für den Teilnehmer `fix` in der Paßwortdatei `/etc/passwd`.

```
fix:R9VtMw:201:50::/usr/fix
```

Professor Fix hat also

- den *Teilnehmernamen* (login-Name) `fix`,
- die *Teilnehmerkennung* (user-id, `uid`) `201`,
- die *Gruppenkennung* (group-id, `gid`) `50`.

Die Kennungen werden vom Systemverwalter (meist fortlaufend) vergeben. UNIX verwendet diese Nummern anstelle von Namen zur internen Verwaltung seiner Aufgaben.

In der Paßwortdatei sind die *Gruppennamen* nicht genannt. Die Zuordnung zwischen der Gruppenkennung (`gid`) und dem Gruppennamen wird über die Datei `/etc/group` hergestellt, die wie `/etc/passwd` nur vom super-user geändert werden darf.

Im Rechnersystem von Professor Fix sind alle Teilnehmer automatisch Mitglied in der Gruppe `group` (`gid 50`), andere Systeme haben andere Namen für die Defaultgruppe.

```
$cat /etc/group
root:x:0:root
cron:x:1:cron
bin:x:3:bin
uucp:x:4:uucp
asg:x:6:asg
group::50:fix,freundl,jr,dekan,neu
fbmi::51:dekan,fix,jr,freundl
stkom::52:dekan,fix,jr,neu
$
```

Wie Sie sehen, ist Professor Fix auch noch Mitglied in der Studienkommission (`stkom`, `gid 52`) und im Fachbereich (`fbmi`, `gid 51`). Kehren wir nun mit Professor Fix zu der Datei `PrO96` (Prüfungsordnung) zurück.

```
$echo Entwurf Pruefungsordnung >PrO96
$ls -lg PrO96
-rw-r--r-- 1 group    25 Feb 16 PrO96
$
```

Der momentane Besitzer ist `fix`, die besitzende Gruppe ist die login-Gruppe `group`. Unser Ziel soll es nun sein, den Dekan zum Besitzer zu machen und der Studienkommission (`stkom`) das Schreibrecht einzuräumen.

```
$chgrp stkom PrO96
$chmod g+w PrO96
$chown dekan PrO96
$
```

Die einzelnen Schritte sind:

- Die Besitzergruppe mit `chgrp` *gruppenname datei* ... ändern.
- Der Gruppe das Schreibrecht einräumen.
- Den Besitzer mit `chown` *teilnehmername datei* ... ändern.[1]
- Das Werk stolz betrachten:

```
$ls -l PrO96
-rw-rw-r-- 1 dekan   25 Feb 16 PrO96
$ls -lg PrO96
-rw-rw-r-- 1 stkom   25 Feb 16 PrO96
```

1. In neueren UNIX-Versionen ist `chown` für alle außer dem super-user gesperrt.

Da Professor Fix Mitglied der Gruppe `stkom` ist, kann er also weiterhin `PrO96` bearbeiten. Oder doch nicht?

```
$echo Erste Sitzung 3.Maerz >>PrO96
PrO96: cannot create
$
```

Das Problem ist die falsche momentane Gruppenkennung (`group` aus der Anmeldung). Mit `newgrp` (log into a **new gr**oup) ändert Professor Fix seine Gruppenkennung für diese Sitzung,

```
$newgrp stkom
$echo Erste Sitzung 3.Maerz >>PrO96
$
```

und jetzt klappt es!

```
$cat PrO96
Entwurf Pruefungsordnung
Erste Sitzung 3.Maerz
$
```

Nachträglich möchte er jetzt auch noch allen anderen Benutzern das Lese- und Schreibrecht für die Datei `PrO96` einräumen.

```
$chmod o=rw PrO96
chmod: can't change PrO96
$
```

Dies geht nicht mehr, denn `fix` ist nicht mehr der Besitzer der Datei und die Gruppenmitgliedschaft allein genügt nicht. Mit `su` (**su**bstitute user id temporarily) könnte er kurz die Identität des Dekans annehmen,

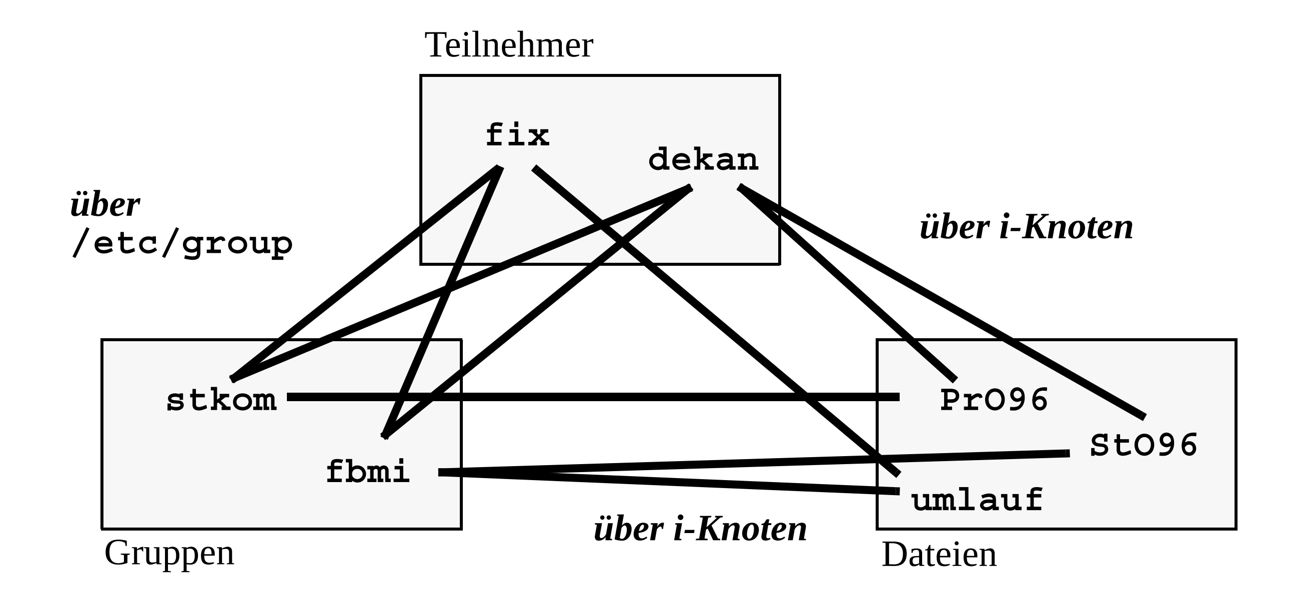
```
$su dekan
Password:.......
Password incorrect
$
```

aber er muß passen.

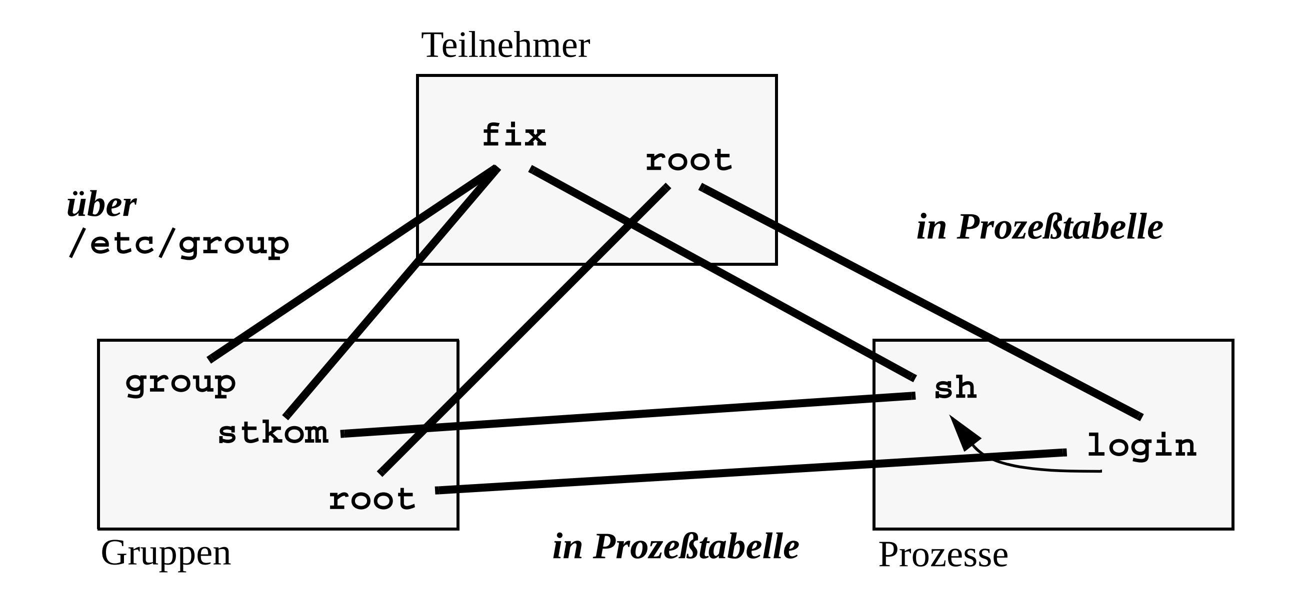
Aus obigen Dialogen wird klar, daß jede Datei zur Wahrung der Zugriffsrechte die Kennung des Besitzers (*owner-user-id*) und die Kennung der Besitzergruppe (*owner-group-id*) mitführt. Diese Besitzerinformation steht zusammen mit den Rechten in dem *i-Knoten* der Datei.

Teilnehmer, Gruppen und Dateien bilden zusammen eine Dreiecksbeziehung, die bis auf `chown`, `chgrp` und Änderungen in `/etc/group` prinzipiell statischer Art ist.

Eine weitere Dreierbeziehung gibt es zwischen Teilnehmern, Gruppen und *Prozessen*, denn auch Prozesse haben Besitzer. Das Kommando `newgrp` beeinflußt diese Beziehung, in dem es der Shell mitteilt, daß sie ab sofort mit einer neuen Gruppe als Besitzer läuft. Auch die Teilnehmerkennung kann während einer Sitzung gewechselt werden (Kommando `su`).



Insgesamt ist die im zweiten Bild dargestellte Beziehung dynamischer Art, denn Prozesse verzweigen sich (`fork`), vererben ihren Besitzer und können die Kennung anderer Teilnehmer und Gruppen annehmen, wie für den Prozeß `login` des Teilnehmers `root` im Bild angedeutet, der eine Shell erzeugt, die dem angemeldeten Teilnehmer (hier `fix`) übergeben wird.



Die letztgenannte Eigenschaft wird in UNIX auch genutzt, um gewöhnlichen Sterblichen kurzzeitig die Identität des super-users zu geben. Warum und wie? Schauen Sie sich die folgenden zwei Abschnitte an!

Zusammenfassung

❖ Wir haben die *Teilnehmer*- und *Gruppenkennung* (`uid`, `gid`), *Teilnehmer* und *Gruppe* als *Besitzer* von Dateien und Prozessen und deren Beziehungen untereinander kennengelernt. Angewandt wurden die Kommandos `chown`, `chgrp`, `newgrp` und `su`.

Frage 1

In manchen Systemen gibt es ein Kommando `id`, das die momentane Benutzer- und Gruppenzugehörigkeit des aufrufenden Prozesses liefert. Wie gesehen, läßt sich die Gruppenkennung auch abfragen mit

❏ `who -g`

❏ `ps`

❏ `newgrp`

❏ keinem der drei Kommandos.

Frage 2

Welche Aussage(n) ist (sind) **falsch**?

Zu jedem Zeitpunkt kann

❏ ein Teilnehmer Besitzer vieler Dateien sein.

❏ eine Gruppe Besitzer vieler Dateien sein.

❏ eine Datei viele Teilnehmer als Besitzer haben.

❏ eine Datei viele Gruppen als Besitzer haben.

❏ eine Gruppe viele Teilnehmer haben.

❏ ein Teilnehmer Mitglied vieler Gruppen sein.

7.2 Privilegien

❖ In diesem Abschnitt lernen wir, unabhängig von UNIX, das Konzept des privilegierten Zustands und die Methode des kontrollierten Übergangs aus dem Anwenderstatus in den Systemstatus kennen.

Rechenanlagen für Mehrbenutzerbetrieb erzwingen eine Zweiklassengesellschaft.

Anwenderstatus	Systemstatus
keine privilegierten Befehle	privilegierte Befehle
kein Zugang zu Systemtabellen und Geräten	Zugang zu Systemtabellen und Geräten

Diese Trennung in *Anwender*- und *Systemstatus* wird zum Teil auch von der Hardware unterstützt.

Mit einer völligen Trennung wären aber auch alle Dienstleistungen des Systems, z. B. der Druck-Spooler, für Anwender nicht verfügbar.

Anwenderprogramme

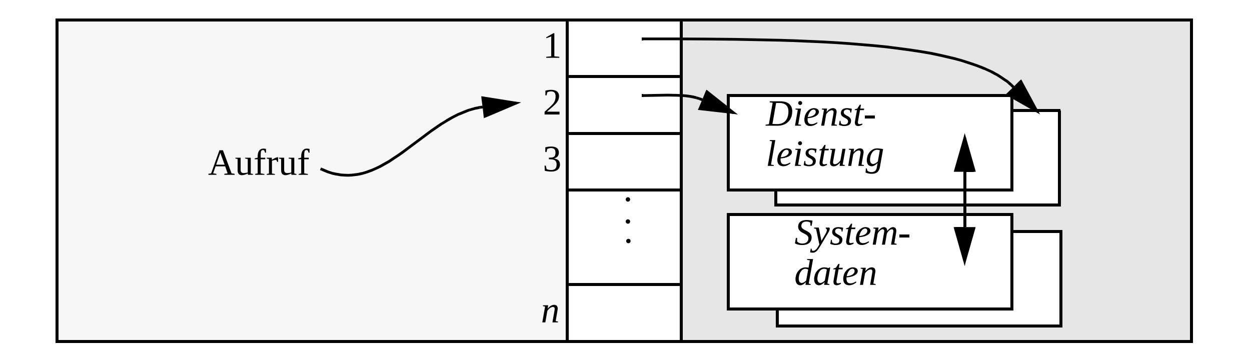
Aufruf von Dienstleistung x?

Systemprogramme

kein Zutritt
für jedermann!

Deshalb schafft man einen wohldefinierten Übergang zwischen Anwendung und System in Form einer Übergangstabelle.

Die Dienstleistungen können jetzt durch indirekten Aufruf in Anspruch genommen werden.

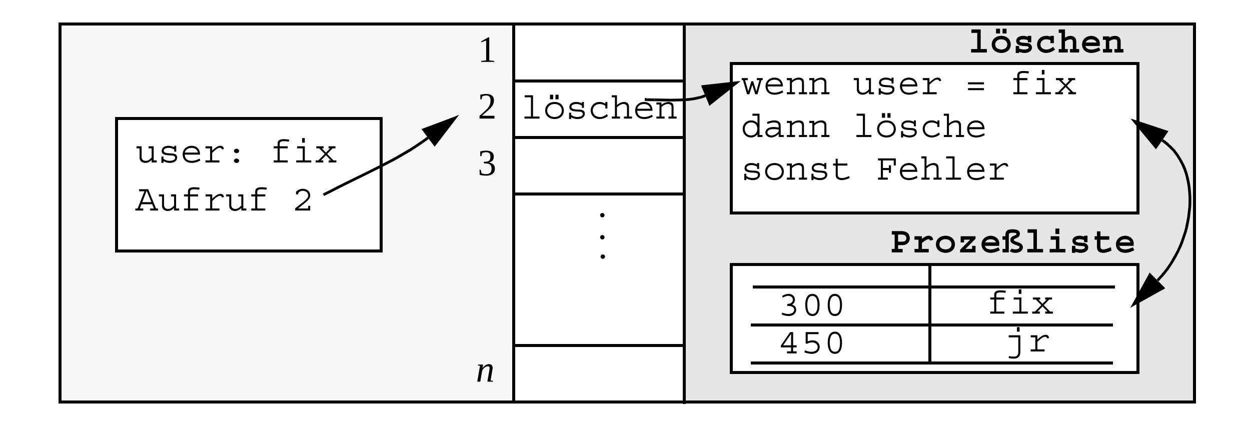


Die entscheidenden Punkte sind:

- Dienstleistungsprogramme und Systemdaten liegen im geschützten Bereich.
- Die Übergangstabelle selbst ist auch geschützt.
- Das Dienstleistungsprogramm kann die Berechtigung des Aufrufs prüfen und führt seine Aufgaben korrekt aus.
- Mit dem Aufruf erfolgt automatisch der Übergang in den privilegierten Status.
- Der Aufruf selbst ist kein privilegierter Befehl.

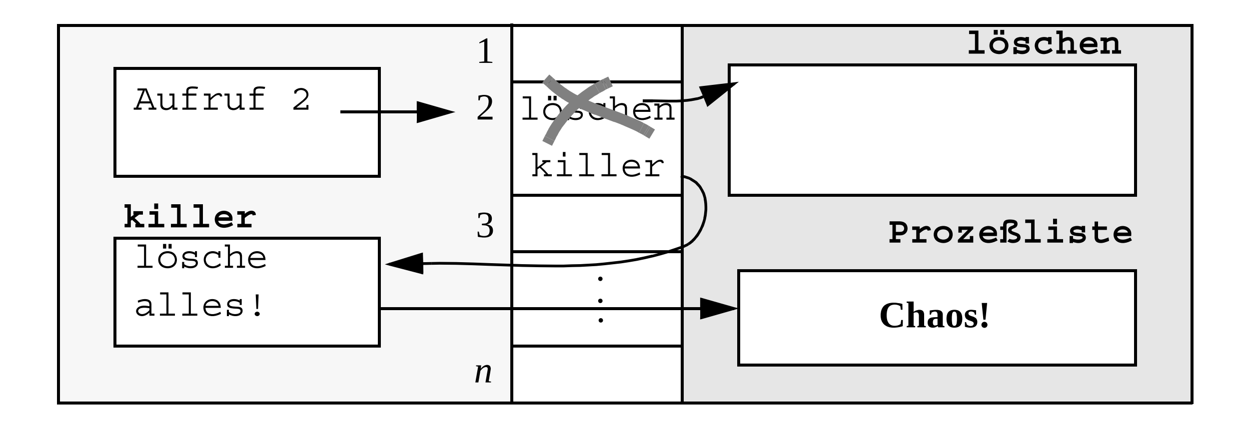
Diese Technik des kontrollierten Übergangs (mit *TRAP*, *Supervisor-Call* oder *Interrupt*) ist seit Jahrzehnten in der Rechnerorganisation bekannt. Sie hat den Vorteil, daß das Angebot an Dienstleistungen zentral über die Einträge in der Tabelle vom Systemverwalter geändert werden kann.

Solange der Tabelleneintrag einer Dienstleistung nicht geändert wird, solange muß auch in den aufrufenden Anwenderprogrammen nichts umgestellt werden, auch nicht bei Änderungen an den Dienstleistungen selbst.



Änderungen in Dienstprogrammen können notwendig werden z. B. zur Anpassung an neue Hardware, bei Überarbeitungen des Betriebssystems oder zur Realisierung von Einschränkungen (während des Übungsbetriebs ist die Textverarbeitung gesperrt).

Wichtig ist, daß sowohl die Tabelle als auch die Dienstprogramme vor Manipulationen geschützt sind, denn mit dem Aufruf erfolgt automatisch ein Übergang in den privilegierten Status. An einem Serviceprogramm „Lösche Prozeß“ sollen die möglichen Folgen einer Nichtbeachtung dieser Vorsichtsmaßnahmen gezeigt werden.



In dem Bild ist zwar das Programm `löschen` geschützt, die Tabelle jedoch nicht. Dadurch kann der Aufruf 2 zum Programm `killer` umgeleitet werden. Wesentlich ist, daß durch den Aufruf immer der Übergang in den privilegierten Zustand erfolgt.

Wenn die Tabelle geschützt ist und das Programm `löschen` nicht, kann die Datei `löschen` durch die Datei `killer` ersetzt werden, mit dem gleichen Ergebnis wie im letzten Bild.

Moderne Rechnerarchitekturen unterstützen das Konzept des kontrollierten Übergangs durch die Bildung getrennter Adreßräume sowie durch getrennte Befehls- und Stapelzeiger für den Anwender- und den Systemstatus. Damit kann ein Prozeß in zwei Phasen arbeiten, einer ungeschützten *Anwenderphase* und einer geschützten, privilegierten *Systemphase*.

UNIX nutzt dies aus für die sogenannten *Systemaufrufe* (*system calls*). Diese Systemprogramme, die von außen wie C-Funktionen aufgerufen werden, realisieren die kritischen Aufgaben der Kommandos und enthalten alle den oben beschriebenen Übergang in den Systemstatus. Die Abwicklung der Systemaufrufe, die zum Teil in Maschinensprache geschrieben sind, ist Aufgabe des *Betriebssystemkerns* (*kernel*).

Zusammenfassung

❖ Wir haben den Anwender- und den Systemstatus sowie den Aufruf von Dienstleistungen, die den Systemstatus verlangen, kennengelernt.

Frage 3

Kann es einen Aufruf „neue Übergangstabelle laden“ geben?

❏ Ja, prinzipiell von jedem aufrufbar.

❏ Ja, aber nur vom Systemverwalter aufrufbar.

❏ Nein, die Tabelle muß geschützt sein.

Frage 4

Über einen Aufruf in einer Übergangstabelle ist ein Systemprogramm „Entferne Plattendatei“ verfügbar, mit dem prinzipiell jede Datei gelöscht werden kann. Das kann nur gutgehen, wenn dieses Programm

❏ lesegeschützt ist.

❏ nicht für jedermann aufrufbar ist.

❏ die Berechtigung zum Löschen prüft.

❏ alle aufgeführten Punkte erfüllt.

7.3 Der patente Trick

❖ In diesem Abschnitt lernen wir den Unterschied zwischen effektiver und wirklicher Teilnehmer-Kennung und deren Verwendung für privilegierte Operationen kennen.

Im vorherigen Abschnitt sahen wir, daß es zur Implementierung beliebiger Systemdienste genügt, wenn die Hardware einen privilegierten und einen nicht privilegierten Zustand unterscheiden kann, einen Übergang zwischen den Zuständen erlaubt und gewisse Speicherbereiche vor dem Zugriff durch nichtprivilegierte Anwender schützt.

UNIX nutzt diesen — je nach Rechner geringfügig unterschiedlichen — Schutzmechanismus aus zur Implementierung seiner Systemaufrufe (nicht zu verwechseln mit den häufig gleichlautenden Kommandos). Einige dieser Systemaufrufe sind

für die Ein-/ Ausgabe:
`open`, `write`, `read`, `close`, `creat`[1],
für das Dateisystem:
`link`, `unlink`, `access`, `mknod`, `chmod`, `chown`,
und für die Prozeßkontrolle:
`fork`, `exec`, `exit`, `wait`, `chdir`.

Während der Ausführung der Systemaufrufe befindet sich ein Prozeß in der Systemphase und verwendet einen getrennten Datenbereich, sonst in der Anwenderphase.

Wie schon oben erläutert, kann jeder Benutzer die Systemaufrufe benutzen, und sie sind – hoffentlich! – so programmiert, daß ein Mißbrauch ausgeschlossen ist.

Die Systemaufrufe können wiederum in Kommandodateien auftreten, die meist mit den Rechten `rwx--x--x` geschützt sind und einem Besitzer namens `bin` gehören. Der Benutzer `bin` hat oft die Kennung (*uid*) 1 im Gegensatz zum super-user root mit der uid 0. Damit dürfen die Kommandos von allen ausgeführt und gegebenenfalls auch gelesen werden, aber nur `bin` und `root` besitzen das Schreibrecht.

Die Überprüfung der Rechte nimmt der Betriebssystemkern durch den Vergleich folgender Kennungen vor (vgl. den ersten Abschnitt dieser Lektion):

- Besitzerkennung (*owner-user-id*) im i-Knoten der Datei,
- Teilnehmerkennung (user-id) im aufrufenden Prozeß.

Die Teilnehmerkennung vererbt sich bei Prozeßverzweigungen (`fork`), d. h. der Sohnprozeß erbt sie vom Vater. Nur der super-user kann die Teilnehmerkennung auf einen beliebigen neuen Wert mit dem Systemaufruf `setuid` setzen.

Die häufigste Anwendung von setuid geschieht in `login`, das nach erfolgreicher Anmeldung seine Benutzerkennung null aufgibt und die des Teilnehmers aus `/etc/passwd` annimmt, um dann unter der neuen Kennung eine Shell (Kommandointerpreter) für den Teilnehmer zu starten.

UNIX hat diesem Sicherheitssystem noch einen weiteren Trick hinzugefügt, für den Dennis Ritchie das U.S. Patent no. 4,135,240 hält.

- Zur Ausführung einer privilegierten Dienstleistung kann man sich die Kennung des Besitzers der Dienstleistung leihen.

Dazu wird die Teilnehmerkennung der Prozesse aufgespalten in

- *wirkliche* Kennung (*real user-id*),

1. Auf die Frage, was er bei einem Neuentwurf von UNIX anders machen würde, antwortete Ken Thompson einmal: „Ich würde creat mit e schreiben!“

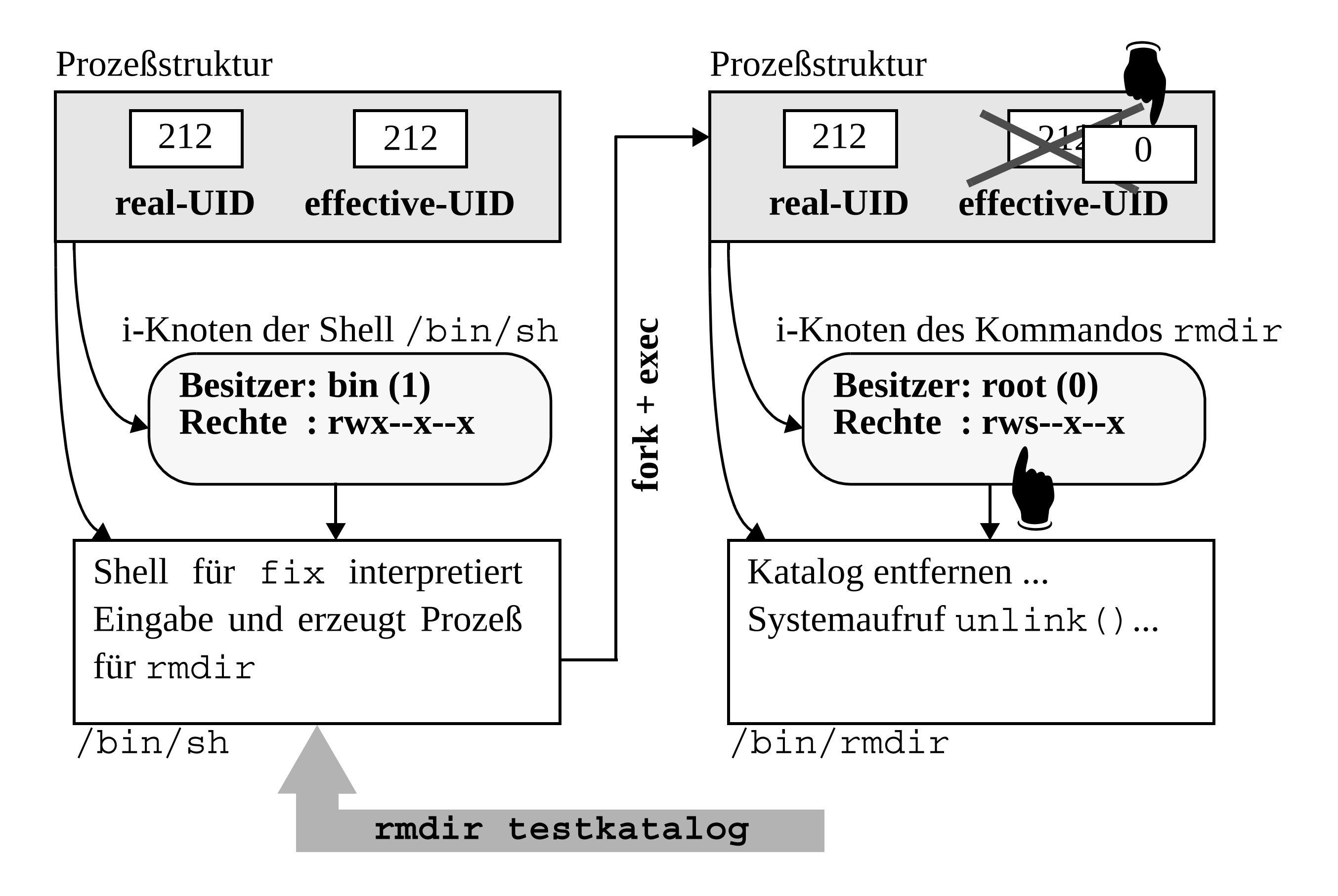
- *effektive* (angenommene) Kennung (*effective user-id*).

Für Gruppenkennungen geschieht die Aufspaltung analog. Ob eine Dienstleistung es zuläßt, daß die effektive Kennung wechselt, bestimmt das *s-Bit* (set user-id) im i-Knoten der Binärdatei, die diese Dienstleistung erbringt. Angezeigt wird sie durch ein `s` statt eines `x` in den Dateirechten.

Hier einige Einträge aus `/usr/bin`, dem Katalog der „binaries“, d. h. der Kommandos, die als direkt ausführbare Maschinenprogramme vorliegen. Mittels `grep` wurde speziell nach Zeilen gesucht, deren s-Bit gesetzt ist.

```
ls -l /usr/bin | grep '^..s'
-r-sr-sr-x 1 root    cron            40700  May 16 2027 at
-r-sr-xr-x 1 root    system         184212  Apr 12 1996 ftp
-r-sr-xr-x 1 root    system           5560  May 16 2027 logout
-r-sr-xr-x 1 root    security        17418  May 16 2027 newgrp
-r-sr-xr-x 1 root    security        11818  May 16 2027 passwd
-r-sr-xr-x 1 root    bin             13112  Apr 26 1996 rlogin
-r-sr-xr-x 3 root    system         179232  Apr 12 1996 telnet
...
```

UNIX verwendet den Trick der gespaltenen Teilnehmerkennung für die Kommandos `at`, `login`, `newgrp`, `passwd`, `rmdir` und `su`, um dem Aufrufer beschränkt die Rechte des Besitzers `root` zu leihen. Die wirkliche Kennung bleibt erhalten und erlaubt weiterhin die Prüfung der Identität des Aufrufers. Das folgende Bild zeigt den vereinfachten schematischen Ablauf beim Wechsel der effektiven Teilnehmerkennung am Beispiel des Kommandos `rmdir`., wobei speziell die Ersetzung der effektiven Benutzerkennung, hier `212` für



Teilnehmer `fix`, durch die Kennung `0` des super-users nach dem `exec` wichtig ist. Die Ersetzung erfolgt aufgrund des s-Bits im i-Knoten des `rmdir`-Kommandos.

Die früher genannten Regeln gelten auch hier wieder:

- Das privilegierte Programm (hier `/bin/rmdir`) und die Systemdateien sind schreibgeschützt.
- Das s-Bit darf nur vom Besitzer (hier `root`) gesetzt werden.
- Das Programm (hier `rmdir`) prüft die Berechtigung der auszuführenden Aufgabe, z. B. nur eigene Kataloge dürfen entfernt werden.
- Währen der Ausführung ist der Prozeß privilegiert (er darf in Kataloge schreiben).
- Jeder darf das Programm aufrufen.

Die bekannteste Anwendung für das s-Bit ist das `passwd` Kommando. Es steht in `/bin/passwd` und manipuliert die Paßwortdatei `/etc/passwd`.

```
$cat /etc/passwd | tail -4
fix:R9VtM:201:50::/usr/fix:/bin/sh
neu:j987Y:202:50::/usr/neu:/bin/sh
jr:j40iHS:203:50::/usr/jr:/bin/sh
dekan:tG6ce:204:50::/usr/dekan:/bin/sh
$ls -l /etc/passwd /bin/passwd
-rws--x--x 1 root    18402 /bin/passwd
-rw-r--r-- 1 root     1324 /etc/passwd
$
```

Die Paßwortdatei ist für jedermann lesbar, denn die Paßworteinträge selbst sind verschlüsselt. Natürlich ist sie schreibgeschützt, außer für den Besitzer `root`.

Damit aber ein normaler Benutzer sein Paßwort ändern kann, leiht der super-user bei der Benutzung des Kommandos `passwd` dem Aufrufer seine user-id als angenommene Kennung.

Systemprogramme mit gesetztem s-Bit müssen sehr sorgfältig konstruiert sein; gelingt es dem Aufrufer eines solchen Programms, in die Shell auszuweichen, ist er praktisch super-user!

Die Verwendung des s-Bits ist aber nicht auf den super-user beschränkt. Jeder Teilnehmer kann ein Programm schreiben, das er selbst besitzt und für dessen Ausführung er seine Teilnehmerkennung verleiht. Professor Fix hat ein ganz einfaches solches Programm erstellt, das einen Briefkasten für die Abgabe von Übungsaufgaben verwaltet.

In `/tmp/fix/ueb` legen die Studenten ihre Lösungen ab und rufen `/usr/fix/abgabe.s` auf, das die Datei an `/usr/fix/mbox` anhängt. Die Datei `mbox` ist für alle außer ihrem Besitzer `fix` lese- und schreibgeschützt.

```
$cat abgabe.s
#Shell-Skript zur Abgabe von Uebungen
```

```
echo Text aus /tmp/fix/ueb wird in \
den Briefkasten kopiert
cat /tmp/fix/ueb >>/usr/fix/mbox
echo und die Datei /tmp/fix/ueb dann gelöscht.
echo -n >/tmp/fix/ueb
$
```

Leider verbietet UNIX diese direkte Lösung. Ein Shell-Skript, wie `abgabe.s` im obigen Dialog, ist kein ausführbarer Code, sondern wird von einer Untershell gelesen. (Trotzdem muß das x-Recht gesetzt sein.)

Der Übergang der effektiven Benutzerkennung von „Aufrufer“ auf „Besitzer der Datei“ bei gesetztem s-Bit erfolgt als Seiteneffekt des Systemaufrufs `exec`, der aber ein ladbares binäres Programm voraussetzt.

Unser cleverer Professor weiß aber einen Ausweg: In einem C-Programm können wir mit dem Systemaufruf `system()` eine Kommandodatei ausführen. Der C-Compiler erzeugt ein ladbares Programm, in unserem Fall `abgabe`, das dann das Shell-Skript `abgabe.s` zur Ausführung bringt.

```
$ls -l abgabe.s abgabe.c abgabe mbox
-rw-r--r-- 1 fix       58 abgabe.c
-rwxr-xr-x 1 fix      181 abgabe.s
-rwsr-xr-x 1 fix     4407 abgabe
-rw------- 1 fix     8730 mbox
$cd /tmp/fix ; ls -l ueb
-rwxrwxrwx 1 fix        0 ueb
$
```

Nur das Programm `abgabe` hat das s-Bit gesetzt. Das C-Programm steht in `abgabe.c`, wir brauchen es nicht zu verheimlichen.

```
$cat /usr/fix/abgabe.c
#include <stdio.h>
main ()
{
 system ("/usr/fix/abgabe.s");
}
$
```

Keine Sorge - dies ist unser einziger Ausflug in die C-Programmierung. Sorge sollte Ihnen jedoch die Sicherheit des Systems machen. Womit wir bei den Schwächen des Schutzkonzepts wären.

Eine mögliche Schwachstelle ist der allmächtige super-user, eine zweite die Dezentralisierung der Übergänge in den privilegierten Status. Dementsprechend wurden über die Jahre hinweg immer wieder Sicherheitslöcher in UNIX entdeckt - und gestopft.

Sind Sie sicher, Professor Fix, daß niemand außer Ihnen einen Schlüssel zu Ihrem Briefkasten besitzt?

`tar` und `mount`: Schau was kommt von draußen rein

Ein abgeschlossenen Rechnersystem läßt sich vor „Einbrüchen“ durch die folgende Regel sichern: Dateien mit Besitzer `root` sind schreibgeschützt, ggf. ist das suid-Bit gesetzt, damit andere an den privilegierten Routinen teilhaben können.

Rechner sind aber nicht abgeschlossen, sie sind vernetzt, haben Laufwerke für entfernbare Datenträger (Disketten, CD-ROM, Streamer, Wechselplatten) oder lassen sich über externe Schnittstellen an vielerlei Geräte anschließen. Auf diesen *Import und Export von Dateien* und die dafür wesentlichen Kommandos — `tar` und `mount` — soll hier kurz eingegangen werden.

Das Kommando `tar` steht für *tape archive*, also die *Archivierung* (Anlegen von Sicherungskopien) von Daten auf Band, wobei Band nicht wörtlich genommen werden darf: Es dürfen auch Platten oder andere Speichermedien sein. Im folgenden Beispiel sichert Professor Fix sein gesamtes Heimatverzeichnis auf dem ersten Diskettenlaufwerk `/dev/fd0` (`fd` für floppy disk). Die Option `c` steht für create, mit `-f` *archiv* gibt man das Archiv, in der Regel ein Gerät, an. Im zweiten Aufruf überprüft Professor Fix dessen Inhalt: `t` für lis**t**, `v` für **v**erbose[1]. Interessant ist auch die Option `z` mit der die Daten komprimiert archiviert, bzw. beim Einladen wieder entpackt werden (vgl. das Kommando `compress`).

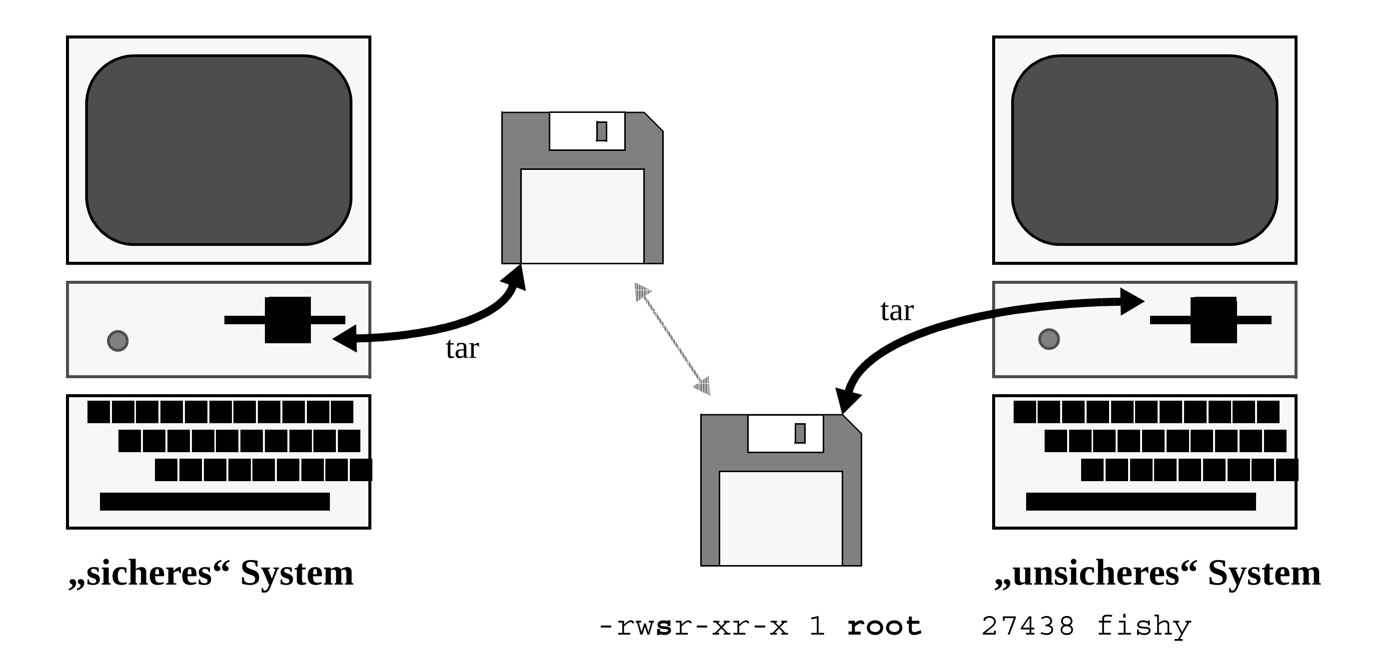
```
$tar -c -f /dev/fd0 /usr/fix
$tar -t -v -f /dev/fd0
drwxr-xr-x 201  50      0 Oct 01 16:46:47 1996 /usr/fix
-rw-r--r-- 201  50    115 Oct 02 16:46:47 1996 /usr/fix/bs
...
```

Die Auflistung zeigt die Benutzer- und Gruppenzugehörigkeit der Dateien und Verzeichnisse. Beim Laden aus einem Archiv (Option `x` für e**x**tract) werden, sofern nicht anders durch Optionen gesteuert, existierende Dateien überschrieben und fehlende im Zielverzeichnis neu angelegt. Wird `tar` vom super-user aufgerufen, kann er existierende Benutzer- und Gruppen IDs des Archivs übernehmen, speziell seine eigenen root IDs, sonst, ohne super-user Berechtigung, werden zwangsweise die des aufrufenden Prozesses genommen.

Damit begegnet UNIX einer offensichtlichen Gefahr, nämlich einer Diskette, die auf einem „feindlichen“ System, etwa einem privaten PC, erzeugt wurde und auf der sich eine root gehörende Shell mit gesetztem suid-Bit befindet (siehe Bild unten). Könnte jedermann diese Datei per Diskette unter dem UID 0 mit gesetztem suid-Bit ins Dateisystem bringen, hätte man eine Shell mit Berechtigung des super-users zur Verfügung.

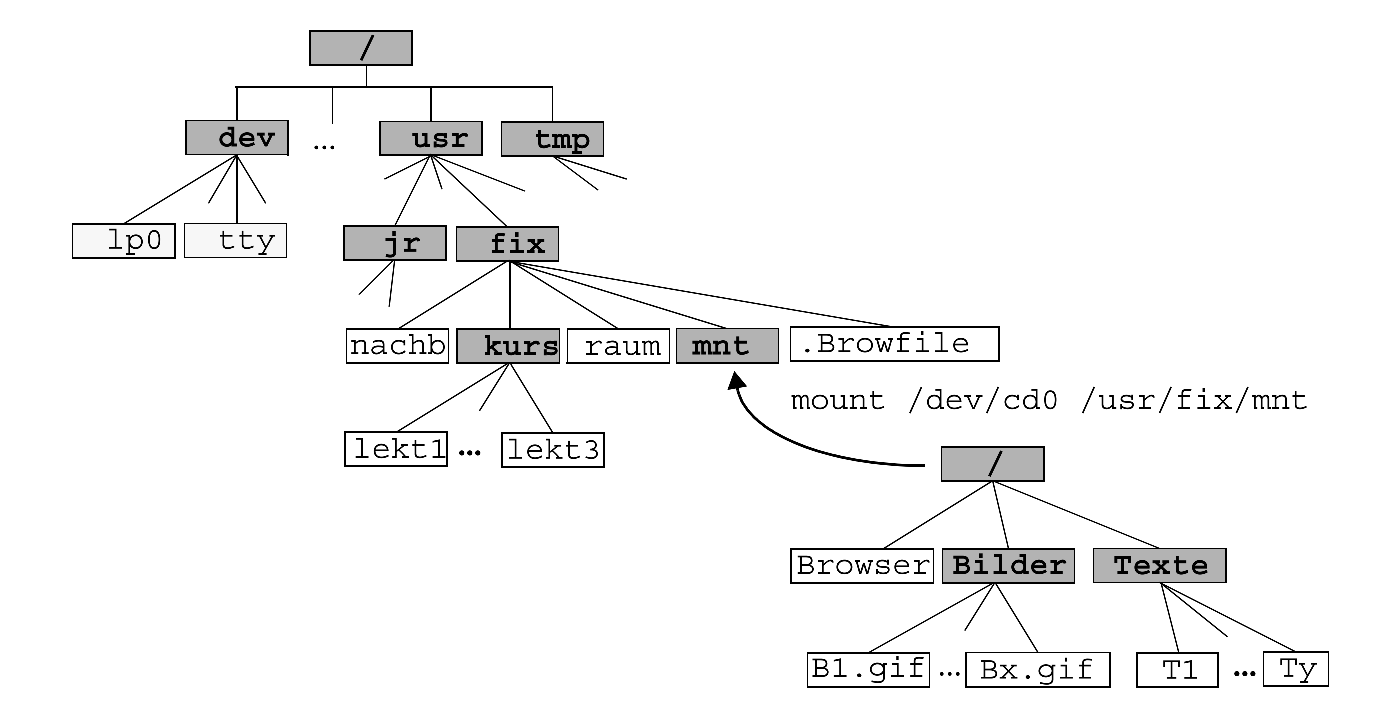
Diskettentransport wird auch gerne als Turnschuhnetz bezeichnet, womit auf die Vorliebe jüngerer Computer-Freaks für bequemes Schuhwerk angespielt wird. Ob per Diskette oder

1. verbose, *engl.* wortreich, geschwätzig, weitschweifig. Eine Option, die bei vielen Kommandos setzbar ist und den Anwender über gestartete Aktivitäten auf dem laufenden hält; sie mildert die für UNIX typische Wortkargheit.



über ein lokales Netz, Dateibäume können nicht nur in existierende Dateisysteme kopiert werden, sie lassen sich auch mittels `mount` „reinhängen“ — „mounten“ oder „montieren“, wie es im Jargon heißt.

Dabei wird ein (vorzugsweise leeres) Verzeichnis im existierenden Dateisystem so mit der Wurzel des montierten Dateibaums verbunden, daß beide nach außen, also besonders für alle Kommandos, wie ein einziges Verzeichnis wirken, unterhalb dessen sich jetzt der neue Baum befindet.



In dem Beispiel im Bild hängt Professor Fix ein UNIX-Dateisystem auf einer CD-ROM in sein Verzeichnis `mnt`. Aus dem Arbeitsverzeichnis heraus könnte er dann den Browser aufrufen mit `mnt/Browser`. Dieser wiederum könnte auf eine Konfigurationsdatei, im Beispiel `.Browfile`, mittels `../.Browfile` zugreifen, vorausgesetzt er hat sein

Arbeitsverzeichnis in der Wurzel des montierten Unterbaums und kann sich darauf verlassen, daß für ihn eine Konfigurationsdatei dieses Namens immer im nächsthöheren Katalog abgelegt wird.

Diese Technik ist Grundlage verteilter Dateisysteme, etwa des NFS (Network File System), die einen transparenten Zugriff auf fremde Dateien bieten. Mit `mount` ohne Argumente erhält man eine Übersicht der montierten Dateisysteme. Diese entstammt einer Tabelle, die der UNIX-Kern unterhält und in der eine Zeile jeweils für ein Pärchen von i-Knoten und Gerätenummern zuständig ist.

Das `mount`-Kommando ist jedoch auf einigen UNIX-Systemen nur vom super-user aufrufbar. Den Grund haben wir bereits oben kennengelernt: das montierte System könnte ein suid-Programm mit Besitzer root enthalten. Im Zeitalter der beschreibbaren CD-ROMs kann man nicht einmal den Silberscheiben trauen — selbstgebrannte CD-ROMs schlimmer als selbstgebrannter Schnaps?

Zuletzt noch ein Ausflug in die Niederungen ganz unaussprechlicher Betriebssysteme. Ist man gezwungen, Daten von, bzw. zu, MS-DOS-Systemen zu übertragen, kann man auf eine mit *mtools* bezeichnete Sammlung von Programmen zurückgreifen, die DOS-Kommandos — jeweils mit dem Präfix `m` — für das Diskettenlaufwerk realisieren. Mit `mcopy` lassen sich Dateien in das UNIX-Dateisystem kopieren:

```
$mdir
 Volume in drive A has no label
 Directory for A:/

README TXT      2965  6-28-95 2:08p
BACKUP DAT   1428137 12-14-95 6:05a
 2 File(s) 25600 bytes free
$mcopy a:README.txt .
Copying README.TXT
$ls *.txt
readme.txt
$
```

DOS- und UNIX-Konventionen für Dateinamen sind nicht immer kompatibel. Daher werden Dateien ggf. umbenannt. Weitere Kommandos sind z. B. `mcd`, `mdel`, `mformat`.

Damit aber genug und immer daran denken: wirkliche Programmierer (REAL programmers) hassen koffeinfreien Kaffee und benutzen keine Disketten!

Zusammenfassung

❖ Wir haben die Rolle der Systemaufrufe und den Unterschied zwischen effektiver und wirklicher Teilnehmerkennung behandelt.

Frage 5

Um über das Kommando `/bin/passwd` die Datei `/etc/passwd` ändern zu können, muß das s-Bit gesetzt sein in

❏ `/etc/passwd`

❏ `/bin/passwd`

❏ `/etc/passwd` und `/bin/passwd`

❏ keiner der genannten Dateien, wenn `root` der Besitzer ist.

Frage 6

Besitzer des Kommandos `rmdir` ist `root`. Das s-Bit ist gesetzt, damit

❏ der Aufrufer jeden beliebigen Katalog löschen kann (Privileg von `root`).

❏ der Aufrufer einen Eintrag in einem Katalog löschen kann (Privileg von `root`).

❏ der Aufrufer beides kann.

Frage 7

Nehmen Sie an, die folgenden Kommandozeilen würden akzeptiert!

```
cat /bin/sh >troja # kopieren
chmod u=sx troja # s-Bit setzen
chown root troja # dem super-user die Shell schenken
```

❏ Durch Aufruf von `troja` hätten Sie eine super-user Shell.

❏ Die Kommandodatei ist ungefährlich, da `troja` nur das s-Bit gesetzt hat.

❏ Die Kommandodatei ist ungefährlich, da `root` Besitzer von `troja` ist.

LEKTION 8:
Benutzung der Shell

8.1 Ganz prompt

❖ In diesem Abschnitt beginnen wir, die Interpretation von Kommandos durch die Shell zu behandeln. Wir unterscheiden einfache Kommandos, Pipelines und Listen von Kommandos und betrachten die Abarbeitung durch eine Untershell.

Zwei Eigenschaften zeichnen gutes Hauspersonal aus: diskretes Auftreten und stete Dienstbereitschaft. Insofern ist der UNIX Kommandointerpreter, also die Shell, der perfekte dienstbare Geist. Außer einem bescheidenen Dollarzeichen und gelegentlich einem knappen, aber bestimmten `cannot create`, `cannot execute` usw. ist wenig zu sehen.

Dabei nimmt die Shell ständig Kommandos entgegen, ersetzt Metazeichen, wie z. B. den Stern und das Fragezeichen, ruft die entsprechenden Programme auf, erledigt kleinere Arbeiten wie `cd` selbst, lenkt die Ein- und Ausgabe um, leitet Pipes und Hintergrundverarbeitung ein, rechnet die Zeit beauftragter Prozesse ab, usw. Und die Shell ist anpassungsfähig.

```
$PS1='Ja bitte? '
Ja bitte? date
Mon Mar 24 15:05 GMT 1997
Ja bitte?
```

Hinter `PS1` (primäres Promptsymbol) verbirgt sich eine Variable der Shell, deren Wert jederzeit neu besetzbar ist. Für den super-user ist die Vorbesetzung ein Doppelkreuz (#), um seine Sonderrolle hervorzuheben. Wir bleiben aber doch lieber beim Dollarzeichen als Promptsymbol.

```
Ja bitte? PS1=$
$
```

Die Hauptaufgabe der Shell ist es, Kommandos zu erkennen und die aufgerufenen Kommandos mit Argumenten zu versorgen. Hier einige einfache Kommandos, die Ihnen vermutlich mehr Nachdenken verursachen als der Shell.

```
$echo Hallo Leute >echo
$wc echo
 1 2 12 echo
$echo wc
wc
$
```

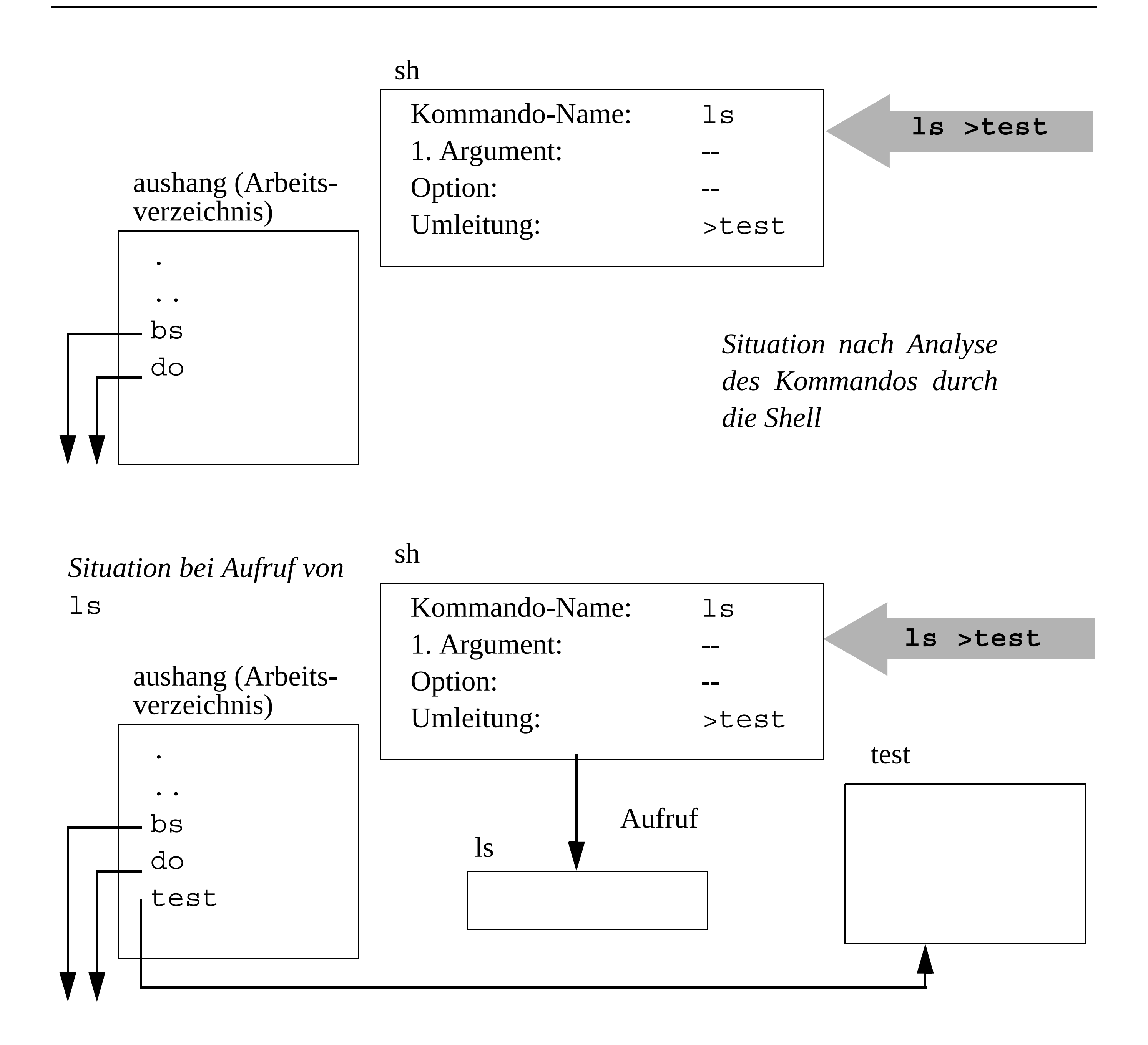
Ein *einfaches Kommando* ist eine Folge von Worten, getrennt durch Tabulator- oder Leerzeichen, den sog. *white spaces* (vgl. Abschnitt 3.1). Das erste Wort wird als *Kommandoname* angesehen. Ferner erkennt die Shell die *Metazeichen* `;` `&` `(` `)` | ^ *Zeilenende Zwischenraum* und *Tabulator* in einer Kommandozeile als Wortbegrenzer.

Nicht zum Kommando selbst gehören die Angaben zur Ein- und Ausgabeumlenkung, wohl aber die Optionen. Damit hat die Shell auch bei unkonventionellen Zeilen (siehe unten) keine Schwierigkeiten.

Das Anlegen, Öffnen und gegebenenfalls Löschen der mit <, > und >> angegebenen Dateien erfolgt vor dem Aufruf der Kommandos. Hier einige der ungewöhnlichen Zeilen.

```
$>echo echo -n wc echo
$cat echo
wc echo$echo echo                 ← keine neue Zeile wegen Option -n
echo
$>echo echo wc echo -n
$cat echo
wc echo -n
$-n echo Hallo Leute!
-n: not found
$
```

Wie man sieht, wird eine Optionsangabe nur unmittelbar nach dem Kommandonamen, der am weitesten links steht, erkannt.

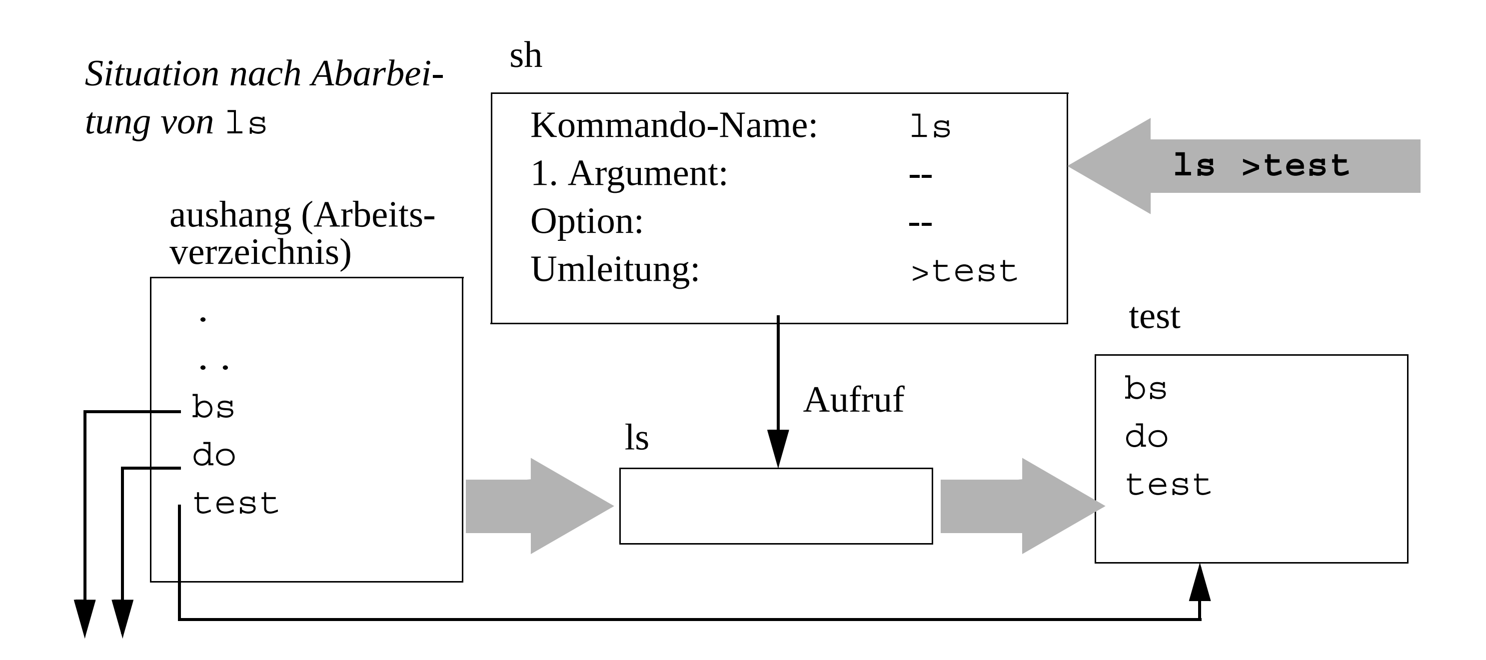


Einfache Kommandos lassen sich mit einem senkrechten Strich (|) oder auch mit einem Dach (^) zu *Pipelines* verbinden. Bei der Abarbeitung einer Pipeline wartet die Shell auf die Beendigung des letzten Kommandos in der Pipe.

Mit einfachen Kommandos und Pipelines lassen sich *Listen* bilden, wobei das Semikolon (`;`) und Ampersand (`&`) *Trenner* sind, die eine sequentielle Abarbeitung bzw. eine *Abarbeitung im Hintergrund* veranlassen. Es dürfen auch beliebig viele Zeilenende-Zeichen in der Liste zum Trennen der Kommandos stehen. Wir haben das beim Verfassen von Shell-Skripten schon ausgenutzt.

Weitere Interpretationsaufgaben der Shell demonstrieren die folgenden Dialoge.

```
$cat mkfile
#Anlegen einer leeren Datei
echo -n Neue Datei $1 in Katalog" "
```




```
pwd ; echo -n >$1
$mkfile test
Neue Datei test in Katalog /usr/fix
$wc test
 0 0 0 test
$
```

Das Shell-Skript `mkfile` zeigt noch eine weitere wichtige Aufgabe der Shell, nämlich die Substitution des Parameters `$1` durch das erste Argument, hier `test`. Die Ersetzung erfolgt zur Laufzeit, d. h. wenn `mkfile` aufgerufen wird. Hier einige weitere Beispiele für die Interpretation von Kommandozeilen durch die Shell.

```
$rm test
$l2
aushang                 raum
mkfile                  studord
nachb3
$cd aushang ; ls
bs
do
$
```

Der Aufruf von `cd` und `ls` in einer Zeile unterscheidet sich nicht vom Aufruf in zwei Zeilen, d. h. das Kommando `ls` beginnt nach wie vor erst dann zu arbeiten, wenn `cd` fertig ist, aber es erscheint nur ein Promptzeichen.

```
$pwd ; ls | wc
/usr/fix/aushang
 2 2 6
$cd ; pwd ; ls | wc
/usr/fix
 5 5 35
```

```
$
```

Hier wurde nur die Ausgabe von `ls` durch `wc` gezählt,

```
$(pwd ; ls) | wc
 6 6 44
$
```

während bei Klammerung die gesamte Ausgabe umgeleitet wird.

```
$(ps ; date) ; ps
PID  TTY     TIME    COMMAND
 79  co      0:05    sh
 90  co      0:00    sh
 91  co      0.03    ps
Mon Mar 24 14:15:56 GMT 1997
PID  TTY     TIME    COMMAND
 79  co      0:05    sh
 92  co      0:03    ps
$
```

Kommandos in Klammern und Shell-Skripte werden in einer eigenen *Untershell* ausgeführt. Deshalb taucht `sh` in der ersten Prozeßliste auch zweimal auf.

```
$cd aushang
$(cd .. ; mkfile test) ; pwd
Neue Datei test in Katalog /usr/fix
/usr/fix/aushang
$(cd .. ; rm test ; l2 ; pwd) ; pwd
aushang             raum
mkfile              studord
nachb3
/usr/fix
/usr/fix/aushang
$
```

Änderungen von Variablenwerten in einer Untershell, z. B. die Änderung des Zeigers auf den aktuellen Katalog durch `cd`, haben nur lokale Wirkung. Nach der Rückkehr aus der Untershell gilt wieder der alte Wert. Das macht deutlich, daß UNIX kein Konzept einer Terminalsitzung mit zugeordnetem Arbeitsverzeichnis kennt.

Zusammenfassung

❖ Wir haben die Interpretation von Kommandos durch die Shell und die Gruppierung von Kommandos durch Klammern, Semikolon, Pipesymbol und Ampersand (Hintergrundverarbeitung) kennengelernt.

Frage 1

Welche Größe wird für die Datei `test` angezeigt?

```
$ls -l >test
$cat test
drwxr-xr-x 2 fix              80 Feb 16 aushang
 ...
-rw-r--r-- 1 fix               ? Feb 25 test
$ls -l
drwxr-xr-x 2 fix              80 Feb 16 aushang
 ...
-rw-r--r-- 1 fix             215 Feb 25 test
$
```

Frage 2

Sie hätten bei der Eingabe einer Kommandozeile lieber automatisch eine Leerstelle zwischen dem Promptzeichen `$` und dem Kommandonamen, also z. B. `$ date` statt `$date`. Um das zu erreichen, müssen Sie

❏ `PS1="$ "` setzen,

❏ mal ein ernstes Wort mit dem super-user reden,

❏ sich wohl oder übel einen neuen Rechner kaufen.

Frage 3

Das Kommando `echo Alles machbar, Herr Nachbar >-n` liefert

❏ die Ausgabe (keine neue Zeile)
`Alles machbar, Herr Nachbar >$`

❏ eine Fehlermeldung

❏ eine Datei namens `-n` mit Inhalt
`Alles machbar, Herr Nachbar`

8.2 Muster mit Wert

❖ In diesem Abschnitt lernen wir den Ersetzungsmechanismus der Shell zur Erzeugung von Dateinamen aus Mustern und das Prinzip der Fluchtsymbole zur Unterdrückung der Ersetzung kennen.

In manchen Kartenspielen gibt es Karten, die man immer spielen kann. Im Englischen heißen sie *wildcards*. In Anlehnung daran werden Metazeichen (`*`, `?`, `[...]`) in Shell-Kom-

mandos als *wildcard Symbole* bezeichnet, denn sie produzieren Argumente, die immer passen.

Erkennt die Shell in einem Wort einer Kommandozeile ein Metazeichen, betrachtet sie das Wort als Muster. Ein Muster wird in der Sortierfolge nacheinander durch alle passenden Dateinamen des Arbeitsverzeichnisses ersetzt. Das schafft die Möglichkeit, Dateinamen abzukürzen. Im nächsten Beispiel werden die Zeilen der verschiedenen Studienordnungen gezählt.

```
$cd studord
$l2
StO97.1             StO97.3
StO97.2             status
$wc -l StO97.?
 594 StO97.1
 667 StO97.2
 667 StO97.3
1928 total
$
```

Eine passende Ersetzung für ein Fragezeichen `?` muß ein einzelnes Zeichen sein, passende Ersetzungen für einen Stern `*` sind beliebige (auch leere) Zeichenketten.

```
$wc -l S* | pr -3t
594 StO97.1         667 StO97.3             1928 total
667 StO97.2
$
```

Mit Hilfe des Sterns wird der Aufruf von `wc` noch kürzer. Und was bewirkt ein Stern allein?

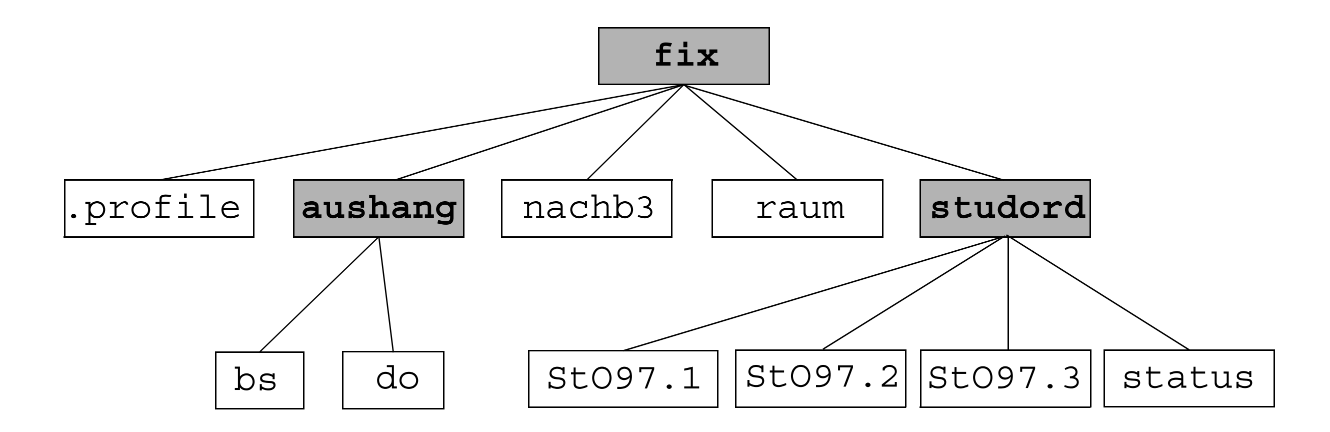
```
$wc -l *
 594 StO97.1
 667 StO97.2
 667 StO97.3
   5 status
1933 total
$
```

Er liefert alle Namen des aktuellen Katalogs, oder fehlen etwa einige?

Mit dem Stern allein erhält man alle Namen des aktuellen Katalogs außer jenen, die mit einem Punkt (`.`) anfangen. Auch ein Schrägstrich (`/`) muß ausdrücklich angegeben werden, d. h. die Ersetzung erfolgt nur für eine Komponente des Pfadnamens.

Das Kommando `echo` ist ideal zum Erkunden des Ersetzungsmechanismus, denn es gibt einfach alle Argumente aus, die es von der Shell nach deren Substitution erhält.

```
$echo *
StO97.1 StO97.2 StO97.3 status
$
```



Das war `echo *` als schwacher Ersatz für `ls`. Aber es geht ja weiter.

```
$echo .*
. ..                         ← alle Verzeichnisse in studord
$cd                            die mit . anfangen
$pwd
/usr/fix
$echo * *
aushang nachb3 raum studord aushang nachb3 raum studord
$echo */*
aushang/bs aushang/do studord/St097.1 studord/St097.2 studor
d/St097.3 studord/status
$ls Aus*
Aus*: not found
$
```

Wenn keine Ersetzung paßt, gibt die Shell das Muster unverändert an das aufgerufene Kommando weiter.

Wie sieht die Ersetzung für eckige Klammern (`[` ... `]`) aus? Sie muß auf eines der eingeschlossenen Zeichen passen, wobei durch einen Bindestrich (`-`) zwischen zwei Zeichen auch ein Ausschnitt des Alphabets angegeben werden kann. Ist das erste Zeichen nach der öffnenden Klammer ein Ausrufezeichen oder `^`, darf die Ersetzung auf keines der eingeschlossenen Zeichen passen.

```
$echo studord/*[23]
studord/St097.2 studord/St097.3
$cd studord
$wc -l *[!1s]
 667 St097.2
 667 St097.3
1334 total
$
```

Sind die Dateien `St097.2` und `St097.3` gleich? Das Kommando `cmp` (**comp**are) vergleicht zwei Dateien, gibt aber, anders als `diff`, nur die Tatsache der Gleichheit oder

Ungleichheit und gegebenenfalls die Stelle des ersten Unterschieds an. Es ist daher schneller als `diff`.

```
$cmp *[23]
$rm *3
$
```

Kein Unterschied! Also weg mit `St097.3`. Womit wir bei einem wunden Punkt unseres dienstbaren Geistes sind: die Dinge werden sehr wörtlich genommen!

Im Falle der Eingabe `rm * 3` hätte das Leerzeichen zwischen `*` und `3` genügt, um alle Normaldateien und die Datei mit dem Namen `3` (Fehlermeldung `rm: 3 non-existent`) zu entfernen.

Dem Kommando `rm` ist dabei kein Vorwurf zu machen. Es bekommt, wie alle Kommandos, die Wildcards nicht zu sehen, denn die Shell erledigt zuerst die Substitution und ruft dann die Kommandos mit den substituierten Argumenten auf.

```
$cd ../aushang ; ls
bs
do
$echo Alles klar ??
Alles klar bs do
$echo Alles klar??
Alles klar??
$echo Sonne, Mond und * *
Sonne, Mond und bs do bs do
$
```

Weniger gefährliche Überraschungseffekte sind natürlich immer möglich. Will man sie vermeiden, muß man die Wirkung der Metazeichen abschalten: mit einem Gegenstrich (\) für das folgende Einzelzeichen, mit einfachen Anführungszeichen (`'...'`) für den ganzen eingeschlossenen Text. Die doppelten Anführungszeichen tun es meist auch, aber die Shell interpretiert dann immer noch einige Zeichen (`$`, `'...'` und \) innerhalb des Textes (siehe Lektion 9).

```
$echo Sonne, Mond und '* *'
Sonne Mond und * *
$echo Alles klar \?\?
Alles klar ??
$echo "Zwei \
>Zeilen ??"
Zwei Zeilen ??
$echo 'Zwei \
>Zeilen ??'
Zwei \
Zeilen ??
$echo 'recht
>tricky'
```

```
recht
tricky
$ls
bs
do
$echo * ; date
bs do
Mon Mar 24 23:55:02 GMT 1997
$
```

Und was passiert wohl jetzt?

```
$echo -l * \; date
-l bs do ; date
$
```

Haben wir Sie endgültig verwirrt? Wenn Sie im letzten Beispiel die lange Ausgabe von `bs` und `do` erwartet haben, sind Sie reif für eine Pause. Schauen Sie doch mal in `/usr/games` nach einem netten Kartenspiel. Aber bitte: keine Wildcards!

Ganz regulär

Das im Zusammenhang mit `find` in der 5. Lektion erwähnte `grep` (Durchsuchen einer Datei nach einem Muster), sowie die verwandten Kommandos `egrep` und `fgrep`, benutzen den Begriff des *regulären Ausdrucks*, mit dem sich unter Verwendung von Metazeichen, speziell `*` und `?`, Suchmuster angeben lassen. Reguläre Ausdrücke werden auch in den `ed`- und `sed`-Kommandos, etwa für die Kommandozeile im `vi`, gebraucht.

Der Begriff regulärer Ausdruck kommt aus der Theorie der Formalen Sprachen. Die mit einem regulären Ausdruck bezeichneten („passenden“) Suchmuster können recht einfach rekursiv definiert werden.

- Jede, auch leere, Folge von Zeichen (außer Metazeichen) ist ein regulärer Ausdruck und bezeichnet sich selbst.
- Sind p und q reguläre Ausdrücke und P und Q die zugehörigen Mengen von Suchmustern, dann sind auch $(p|q)$, (pq), $(p)^*$, $(p)^+$ $(p)?$ reguläre Ausdrücke, deren Suchmuster sich jeweils wie folgt ergeben:
  - durch Suchmuster aus P **oder** aus Q ($P \cup Q$)
  - durch Suchmuster zuerst aus P **und** darauf folgend aus Q (sog. *Verkettung*, PQ)
  - durch ein null- oder mehrfaches Auftreten der Suchmuster aus P
  - durch ein mindestens einmaliges Auftreten der Suchmuster aus P
  - durch ein einmaliges oder fehlendes Auftreten eines Suchmusters aus P.

  Klammern dürfen weggelassen werden, wenn die Vorrangregeln „$*^+?$ vor Verkettung“ passen.

Daneben definiert UNIX noch eine Reihe weiterer Metazeichen, um die Suche nach speziellen Mustern zu erleichtern:

`.`	steht für ein beliebiges Zeichen,
`^`	Anfang der Zeile,
`$`	Ende der Zeile,
`[...]`	Aufzählung von Zeichen, mit `-` auch Ausschnitt, mit `^` auch Negation, wie es auch für die Ersetzungen der Shell gilt und gerade in diesem Abschnitt besprochen wurde,
`\`	schaltet die Wirkung eines folgenden Metazeichens aus.

Da viele dieser Metazeichen auch für die Shell eine Bedeutung haben, wird man in der Regel reguläre Ausdrücke, z. B. bei Aufruf von `grep`, in einzelne Anführungszeichen einschließen, um die Ersetzung der Metazeichen des Ausdrucks durch die Shell zu verhindern.

Zunächst als Beispiel die unspezifische „Suche“ in der Datei `do`. Das Kommando `grep` listet alle Zeilen mit Treffern auf.

```
$grep '.' do
Ankuendigungen fuer das SS
Datenorganisation (3+1)
fuer Hoerer im Hauptstudium.
Zeit: Do 14-16
Ort: HS 100
gez. Fix
$
```

Als nächstes suchen wir nach einem großen ‘S’ am Ende einer Zeile.

```
$grep 'S$' do
Ankuendigungen fuer das SS
$
```

Die Erweiterung von `grep`, `egrep`, kann auch mehrfach geklammerte Ausdrücke verarbeiten. Werden mehrere Dateien als Argumente aufgeführt, stellen `grep, egrep` und das besonders schnelle `fgrep` (nur für feste Zeichenketten als Suchmuster) den Dateinamen voran, wie bei unserer Suche nach einem Paar von Klammern mit einer Zeichenkette dazwischen.

```
$egrep '\(.*\)' do bs
do:Datenorganisation (3+1)
bs:Betriebssysteme (2+2)
$
```

Mit `egrep` lassen sich auch Ausdrücke bilden, die mit dem Operator „oder“ (`|`) gebildet wurden. Hier ein Aufruf zum Finden aller „Umlaute“ mit Angabe der Zeilennummer.

```
$egrep -n 'ae|oe|ue' do
1:Ankuendigungen fuer das SS
3:fuer Hoerer im Hauptstudium.
$
```

Als nächstes alle nichtleeren Zeilen ohne Ziffern.

```
$egrep '^[^0-9]+$' do
Ankuendigungen fuer das SS
fuer Hoerer im Hauptstudium.
gez. Fix
$
```

Statt `egrep` kann auch `grep -E` aufgerufen werden. Daneben existiert eine große Anzahl an weiteren Optionen.

Die Suche (und ggf. Ersetzung) mit regulären Ausdrücken nennt man auch *Pattern Matching* im Gegensatz zum Ersetzungsmechanismus der Shell für Metazeichen, der als *Globbing* bezeichnet wird. Dieser Begriff geht auf ein Kommando `glob` aus prähistorischen Bourne-Shell Zeiten zurück, das die Metazeichenersetzung vornahm.

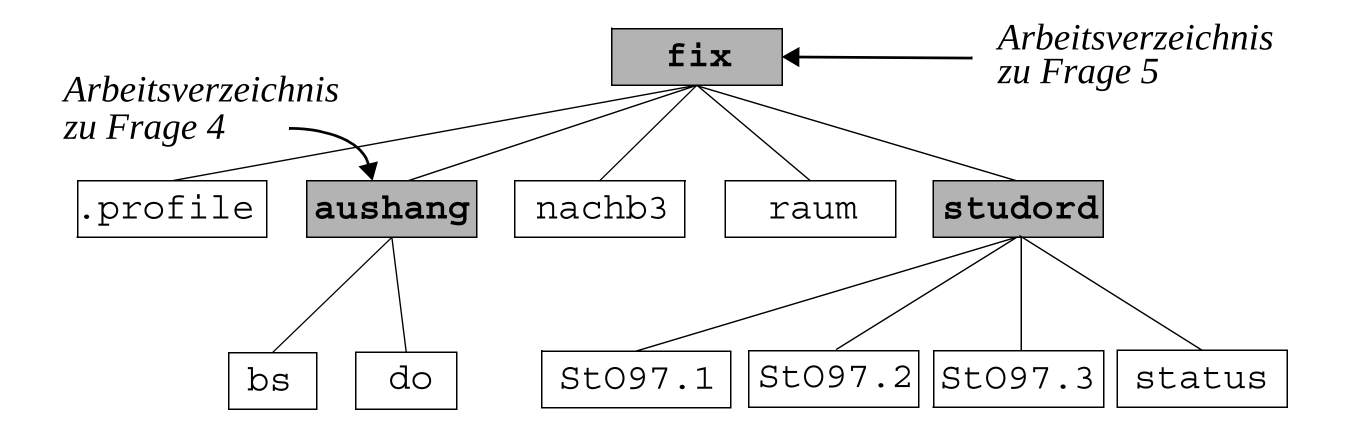
Alles klar? Dann geben Sie [JjYy].* ein!

Zusammenfassung

❖ Wir lernten den Ersetzungsmechanismus der Shell für Muster mit Stern, Fragezeichen und eckigen Klammern und die Aufhebung der Wirkung von Metazeichen durch den Gegenstrich sowie einfache und doppelte Anführungszeichen kennen.

Frage 4

Wieviele Dateinamen werden durch `ls .*` ausgegeben, wenn das Dateisystem wie im Bild unten (Arbeitsverzeichnis `aushang`) aussieht?



Frage 5

Wieviele Dateinamen werden durch `echo [raum]*` ausgegeben, wenn das Dateisystem wie im Bild oben (Arbeitsverzeichnis `fix`) aussieht?

Frage 6

Welche Ausgabe liefert das folgende Kommando?

```
$echo \\, \' und \" sind "Flucht\
>symbole"
```

- ❑ `\\, \' und \" sind Fluchtsymbole`,
- ❑ `\, ' und " sind Fluchtsymbole`,
- ❑ `\, ' und " sind Flucht`
  `symbole`,
- ❑ `\, ' und " sind Flucht\symbole`,
- ❑ keine der angegebenen Ausgaben.

8.3 Selbstversorger

❖ In diesem Abschnitt lernen wir die Versorgung eines Kommandos mit Text aus einem Shell-Skript kennen sowie die bedingte Ausführung von Kommandos. Wir leiten die Standardausgabe und die Standardfehlerausgabe um und verwenden das Kommando `test`.

Die folgende Mitteilung verschickt die Sekretärin des öfteren.

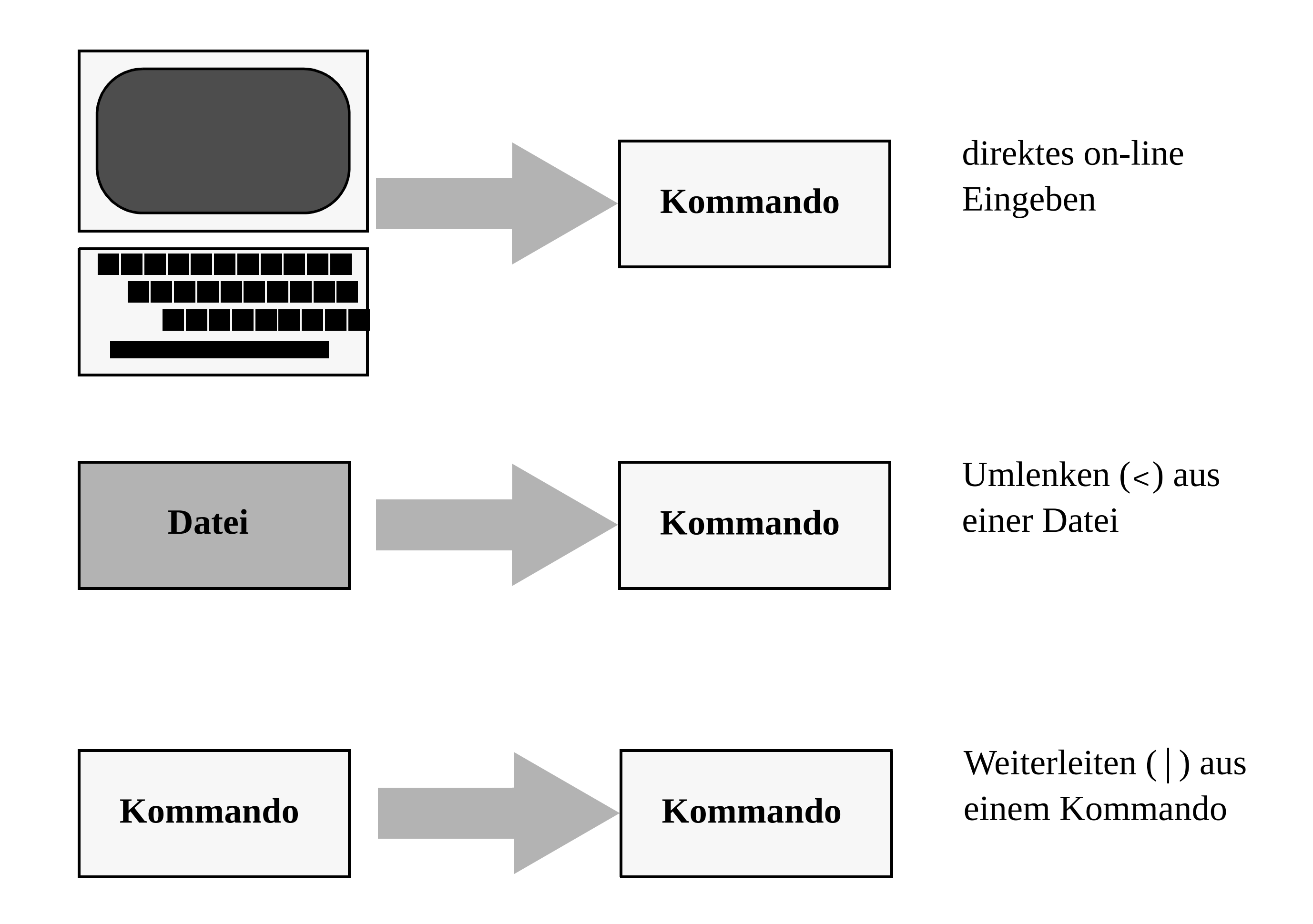
```
You have mail
$mail
From freundl Fri Feb 21 13:13:11 1997
Teilnehmer fix:
Post fuer Sie im Sekretariat.
Gruss Freundlich
?
```

Wie hat wohl die Sekretärin den Text für das `mail` Kommando erstellt? Wir kennen bisher drei Möglichkeiten: Tastatureingabe, Eingabeumlenkung und die Pipe, die alle drei im nächsten Bild noch einmal dargestellt sind.

Für die obige Anwendung im Zusammenhang mit `mail` ist keine der drei Möglichkeiten sehr geeignet.

Sowohl eine Umlenkung der Art `mail fix <notiz` als auch eine Pipe der Form `cat notiz | mail fix` setzen voraus, daß die Datei `notiz` existiert. Soll aber der login-Name des Empfängers in der Notiz erscheinen, muß die Datei `notiz` dazu vorher editiert werden — eine umständliche Lösung.

Bei der Verwendung von `mail fix` ohne Quellenangabe muß der Text jedesmal neu eingegeben werden — eine aufwendige Lösung.



Als Ausweg erstellen wir ein Shell-Skript namens `postda`, das den Rahmentext bereits selbst enthält. Dem Kommando `postda` wird zusätzlich als Argument der login-Name des Empfängers übergeben.

```
$cat postda
(mail $1 <<Ende
Teilnehmer $1:
Post fuer Sie im Sekretariat.
Gruss Freundlich
Ende
) && echo Teilnehmer $1 benachrichtigt
$
```

Das Shell-Skript `postda` enthält natürlich das Kommando `mail`, der Empfänger wird beim Aufruf von `postda` als erstes (und einziges) Argument (`$1`) angegeben. Die Eingabe des `mail` Kommandos steht in `postda` selbst. Es handelt sich hierbei um ein sogenanntes *here-Dokument*, angezeigt durch ein doppeltes Kleinerzeichen (`<<`), gefolgt von dem Namen einer *Endmark*e. Die Eingabe endet vor der Endmarke, hier `Ende`, die separat in einer Zeile stehen muß. Wählt man die Option `<<-`*Marke* werden im here-Dokument führende Whitespaces vor der Marke ignoriert.

Natürlich muß die Datei `postda` auch ausführbar sein.

```
$postda fix
Teilnehmer fix benachrichtigt
```

```
$
```

Im Shell-Skript `postda` wurde ferner eine bedingte Ausführung der Form c_1 `&&` c_2 eingebaut. Dadurch wird das Kommando c_2 nur ausgeführt, wenn das Kommando c_1 fehlerfrei beendet wurde. In unserem Fall erfolgt eine Meldung, wenn `mail` erfolgreich war. Analog wird durch c_1 `||` c_2 das Kommando c_2 nur ausgeführt, wenn c_1 ohne Erfolg blieb.

Die bedingten Ausführungen der Form c_1 `&&` c_2 und c_1 `||` c_2 basieren auf der Tatsache, daß in UNIX jedes Kommando und jeder Systemaufruf als Ergebnis eine Zahl zurückliefert. Dieser *Abschlußstatus* ist Null bei Erfolg und ungleich Null bei einem Mißerfolg. Im letzten Fall zeigt der Wert (Fehlercode) meist zusätzlich noch die Fehlerursache an.

Für Pipelines und Listen von Kommandos ist der Abschlußstatus (*exit status*) des Gesamtausdrucks der Status des letzten ausgeführten einfachen Kommandos, über den Shell-Variablenwert `$?` kann man ihn auch explizit erfragen.

Man beachte, daß die Konventionen in UNIX gegenüber den Darstellungen für Wahrheitswerte in C gerade vertauscht sind.

UNIX		**C**	
= 0	**Erfolg**	**== 0**	**false**
≠ 0	**Fehler**	**!= 0**	**true**

Wir können die bedingte Ausführung auch für unser früheres Kommando `mkfile` einsetzen, um uns zu weigern, eine Datei anzulegen, die bereits existiert. Wir mißbrauchen dazu das Kommando `ls`, um festzustellen, ob die Datei im Arbeitsverzeichnis bereits eingetragen ist.

```
$cat mkfile
#mkfile - 1.Versuch
ls $1 >/dev/null &&
echo Datei $1 existiert bereits
#bei Erfolg von ls eine Meldung
ls $1 >/dev/null ||
(echo -n >$1 ; echo Datei $1 angelegt)
#bei Misserfolg Datei anlegen
$
```

Zur Erinnerung: Die Datei bzw. das Gerät `/dev/null` ist der sog. *Bit-Eimer*. In ihm verschwinden alle eintreffenden Daten spurlos und beim Lesen liefert `/dev/null` stets EOF. Hintergrundprozessen wird das Gerät `/dev/null` als Standardeingabe zugewiesen.

```
$mkfile test
test: not found
test: not found
```

```
Datei test angelegt
$mkfile test
Datei test existiert bereits
$
```

Leider hat der erste Versuch von `mkfile` einen offensichtlichen Nachteil. Existiert die anzulegende Datei nicht, erhalten wir von `ls` über die Standardfehlerausgabe zweimal eine Systemfehlermeldung. Diese wollen wir durch die Umlenkungsangabe `2>/dev/null` (keine Leerzeichen zwischen `2` und `>`) unterdrücken.

Die Syntax dafür bedient sich der sogenannten *Dateideskriptoren*. Das sind ganze Zahlen von 0 an[1], die auf Einträge in einer Systemtabelle (Tabelle der geöffneten Dateien) verweisen. Dateideskriptoren sind an Prozesse gebunden und werden beim Aufruf eines Kommandos in den Systemdatenteil des neuen Prozesses kopiert.

Die drei Deskriptoren

- 0 für die *Standardeingabe*
- 1 für die *Standardausgabe* und
- 2 für die *Standardfehlerausgabe*

werden für jeden Prozeß bereits beim Aufruf belegt. Dabei können mehrere Dateideskriptoren dieselbe Datei bezeichnen, z. B. das eigene Terminal (das ist die Vorbelegung für die Deskriptoren Null bis Zwei).

Die Um- bzw. Zusammenlegung von Deskriptoren wird mit einer syntaktisch unschönen Konstruktion erledigt. Statt `>/dev/null 2>/dev/null` darf man auch `>/dev/null 2>&1` schreiben, und analog wird mit `1>&2` die Standardausgabe zur Standardfehlerausgabe umgeleitet.

```
$cat mkfile
#mkfile - 2. Versuch
ls $1 >/dev/null 2>&1 &&
echo Datei $1 existiert bereits
#bei Erfolg von ls eine Meldung
ls $1 >/dev/null 2>&1 ||
(echo -n $1 ; echo Datei $1 angelegt)
#bei Misserfolg Datei anlegen
$
```

Man beachte, daß die Umlenkung auf die Ausführung des Kommandos `ls` beschränkt ist, `echo` erbt die Vorbelegung von `mkfile`. Auch das neue Kommando `mkfile` gewinnt keinen Schönheitspreis:

- `ls` sollte durch einen richtigen Test ersetzt werden und
- ein schlichtes *if-then-else* würde den Ablauf vereinfachen.

1. für viele Jahre galt 0 bis 19 als typischer voreingestellter Wert

Ersteres läßt sich mit dem Kommando `test` erreichen. Unter den vielen verfügbaren Optionen wählen wir `test -s` *datei*, denn `test` liefert damit den Wert Null, genau dann wenn die angegebene Datei existiert und nicht leer ist. Was jetzt noch fehlt, ist eine einfache Konstruktion für bedingte Kommandos.

```
$cat mkfile
#mkfile - 3. Versuch
if test -s $1
then
 echo Datei $1 existiert bereits
else
 echo -n >$1
 echo Datei $1 angelegt
fi
$
```

Wie man sieht akzeptiert die Shell Kommandos, die wie gewöhnliche Anweisungen in einer höheren Programmiersprache aussehen.

Beachten Sie, das `mkfile` in der jetzigen Form mit `test` die Existenz einer von `mkfile` selbst angelegten Datei nicht erkennt, da sie leer ist. Mehr zur Shell-Programmierung in der nächsten Lektion.

Zusammenfassung

❖ Wir haben den Umgang mit here-Dokumenten (`<<`), die bedingte Ausführung von Kommandos (`&&`, `||`), Dateideskriptoren und das Kommando `test` kennengelernt.

Frage 7

Im Rahmen eines Upgrade auf Release 3.2 lautet `postda` jetzt wie folgt:

```
postda
mail $1 <<##
Fuer $1 :
$2 fuer Sie im Sekretariat
Gruss
##
```

Benachrichtigen Sie mit dem neuen `postda` Kommando den Teilnehmer `jr`, daß ein Paket für ihn eingetroffen ist.

Frage 8

Bei dem Versuch, durch `write` mit einem nicht angemeldeten Teilnehmer einen Dialog zu führen, erscheint eine Fehlermeldung. Wie könnte man in einem privaten Kommando `kwrite` in diesem Fall eine Liste der angemeldeten Teilnehmer erhalten?

- ❏ `write $1 ; who`
- ❏ `write $1 && who`
- ❏ `write $1 || who`

Frage 9

Den Durchfluß einer Pipe können Sie (recht aufwendig) messen, indem Sie mit `tee` eine Kopie erstellen und dann deren Größe ermitteln.

```
$cat meter
tee /tmp/meter$1
wc /tmp/meter$1 ?
rm /tmp/meter$1
```

Wie muß der fehlende Teil heißen, damit das Ergebnis der Zählung auf der Standardfehlerausgabe erscheint?

- ❏ `2>&1`
- ❏ `1>&2`
- ❏ `>/dev/null`

LEKTION 9: Die Parameter der Shell

9.1 Position beziehen

❖ In diesem Abschnitt lernen wir die Positionsparameter der Shell und die Abarbeitung eines Shell-Skripts kennen.

Am Beispiel der selbstverfaßten Kommandos `mkfile` und `postda` haben wir einen ersten Einblick in den *Parametermechanismus* der Shell erhalten. Wir haben gesehen, daß die Shell mit sogenannten *Positionsparametern* `$1`, `$2`, ... arbeitet, die von ihr textuell durch die Argumente des Aufrufs ersetzt werden.

Der Kommandointerpreter gehört daher zu der Klasse der in den späten Sechzigern sehr beliebten *Makroprozessoren*[1]. In den nächsten Bildern zeigen wir vereinfacht das Prinzip des Ablaufs.

1. Die von John K. Ousterhout geschaffene Skriptsprache Tcl (Tool command language) zur Entwicklung graphischer Benutzerschnittstellen ist eine Vertreterin dieser Makrosprachen aus jüngster Zeit.

```
$mkfile test
```

mkfile (Shell-Skript)

```
if test -s $1
then
 echo $1 existiert
else
 > $1
 echo Datei $1 angelegt
fi
```

(a) Das Shell-Skript und sein Aufruf

```
$mkfile test
Datei test angelegt
$
```

mkfile (Shell-Skript)

```
if test -s test
then
 echo test existiert
else
 > test
 echo Datei test angelegt
fi
```

(b) Das Shell-Skript nach der textuellen Ersetzung von `$1` durch `test` und die resultierende Ausgabe

Das Bild (*a*) zeigt die Situation beim Aufruf des Shell-Skripts `mkfile` und den Text des Skripts. In Abbildung (*b*) wurden durch die Shell dann `$1` durch `test` ersetzt und es kommen die entsprechenden (durch Fettdruck markierten) Zeilen durch eine Untershell zur Ausführung. Der Bildschirm bestätigt die erfolgreiche Erzeugung der Datei `test`.

Als nächstes schreiben wir einige Zeichen in die ursprünglich leere Datei `test`, damit wir auch den anderen Zweig der if-then-else-Anweisung probieren können. Den Effekt zeigt Abbildung (*c*) unten, wobei zunächst nochmals eine Datei namens `t2` erzeugt wird, danach erfolgt der erneute Aufruf mit `makefile test`.

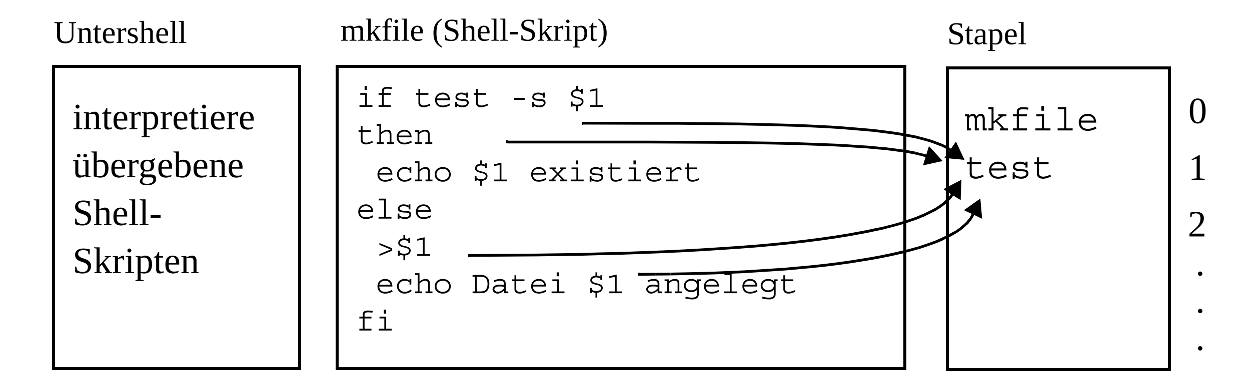
```
$mkfile test
Datei test angelegt
$echo nicht mehr leer >test
$mkfile t2
Datei t2 angelegt
$mkfile test
test existiert
$
```

mkfile (Shell-Skript)

```
if test -s test
then
 echo test existiert
else
 > test
 echo Datei test angelegt
fi
```

(c) Danach aufgerufen mit `t2`, dann nochmals mit `test` und der Hinweis, daß es diese Datei schon gibt.

Die Abarbeitung eines Shell-Skripts wird dadurch realisiert, daß die Parameter (zusammen mit anderen Daten) von der Shell auf dem Laufzeitstapel abgelegt werden, wo sie die Untershell vorfindet, die `mkfile` abarbeitet.



Verdeutlichung der Bindung des Parameters `$1` an das 1. Argument, hier `test` bei Aufruf des Shell-Skripts mit `makefile test`

Man erkennt, daß der Kommandoname als nulltes Argument an die Untershell übergeben wird. Ferner wurde das Kommando `echo -n >$1` durch das leere Kommando ersetzt, d. h. von der Zeile blieb nur die Umlenkung `>$1` übrig.

Einen Makel haben `mkfile` und `postda` allerdings immer noch. Sie können nur ein bzw. zwei Argumente verarbeiten. Das werden wir im nächsten Abschnitt ändern.

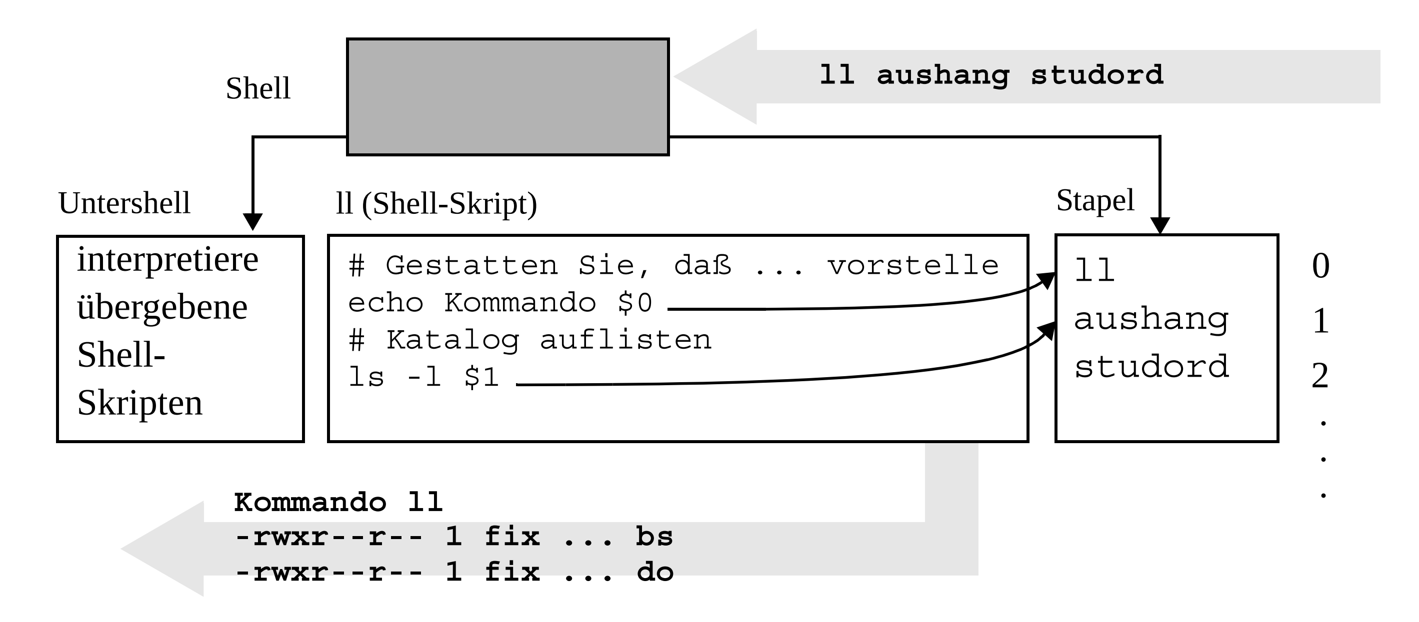
Zunächst betrachten wir ein weiteres selbstverfaßtes Kommando `ll` (**l**ist **l**ong), das (allerdings nicht als Shell-Skript) auf vielen Anlagen verfügbar ist.

```
$cat ll
# Gestatten Sie, daß ich mich vorstelle
echo Kommando $0
# Katalog auflisten
ls -l $1
$
```

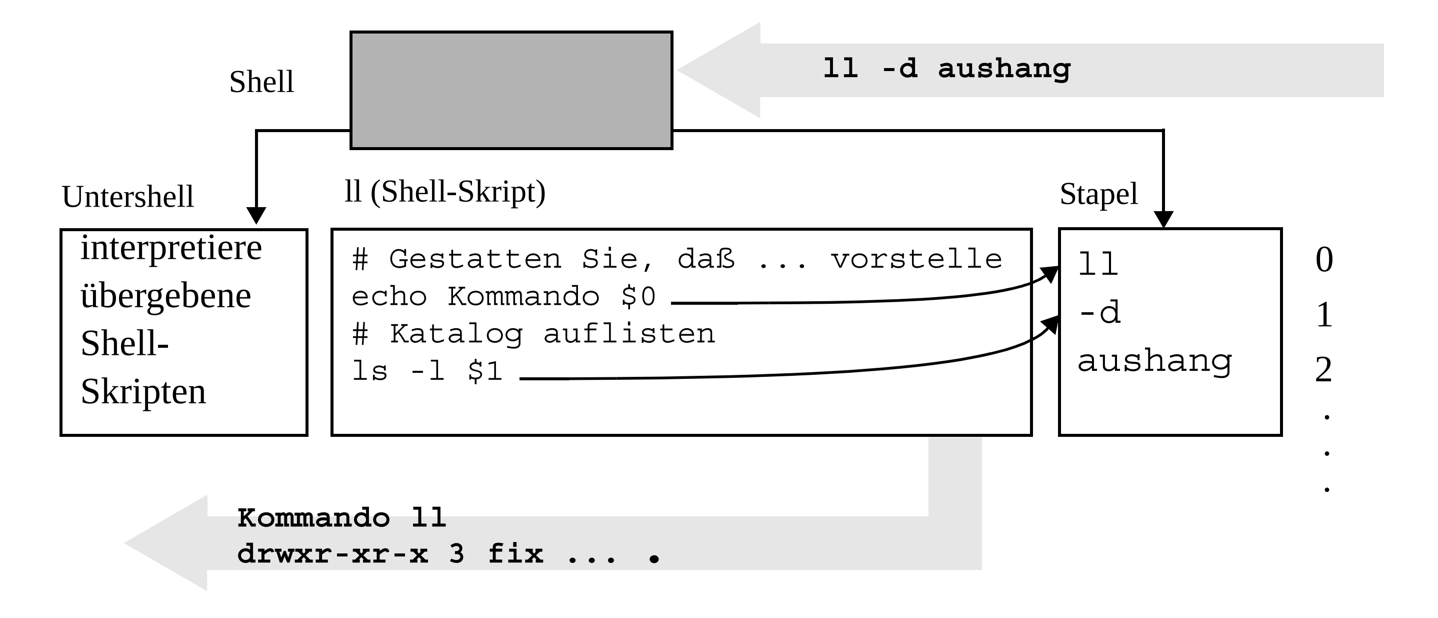
Das Kommando `ll` kann den Inhalt **eines** Katalogs bzw. die Kenndaten **einer** Datei im Langformat ausgeben. Aber auch der Aufruf von `ll` ohne Argumente funktioniert, denn die Shell ersetzt fehlende Argumente durch die leere Zeichenkette `""`, und `ls` liefert dann, wie bekannt, den Inhalt des aktuellen Katalogs.

Wird mehr als ein Dateiname angegeben, wie z. B. bei `ll aushang studord` in der Abbildung unten, gibt die Shell nur `aushang` an `ls` weiter. Das ist gefährlich, weil die fehlende Ausgabe falsch interpretiert werden könnte (z. B. “`studord` ist leer“).

Überraschend — aber konsequent — ist das Ergebnis des Aufrufs `ll -d aushang`. Zur Erinnerung: die Option `d` liefert die Kenndaten des Katalogs selbst, nicht die seiner Einträge.

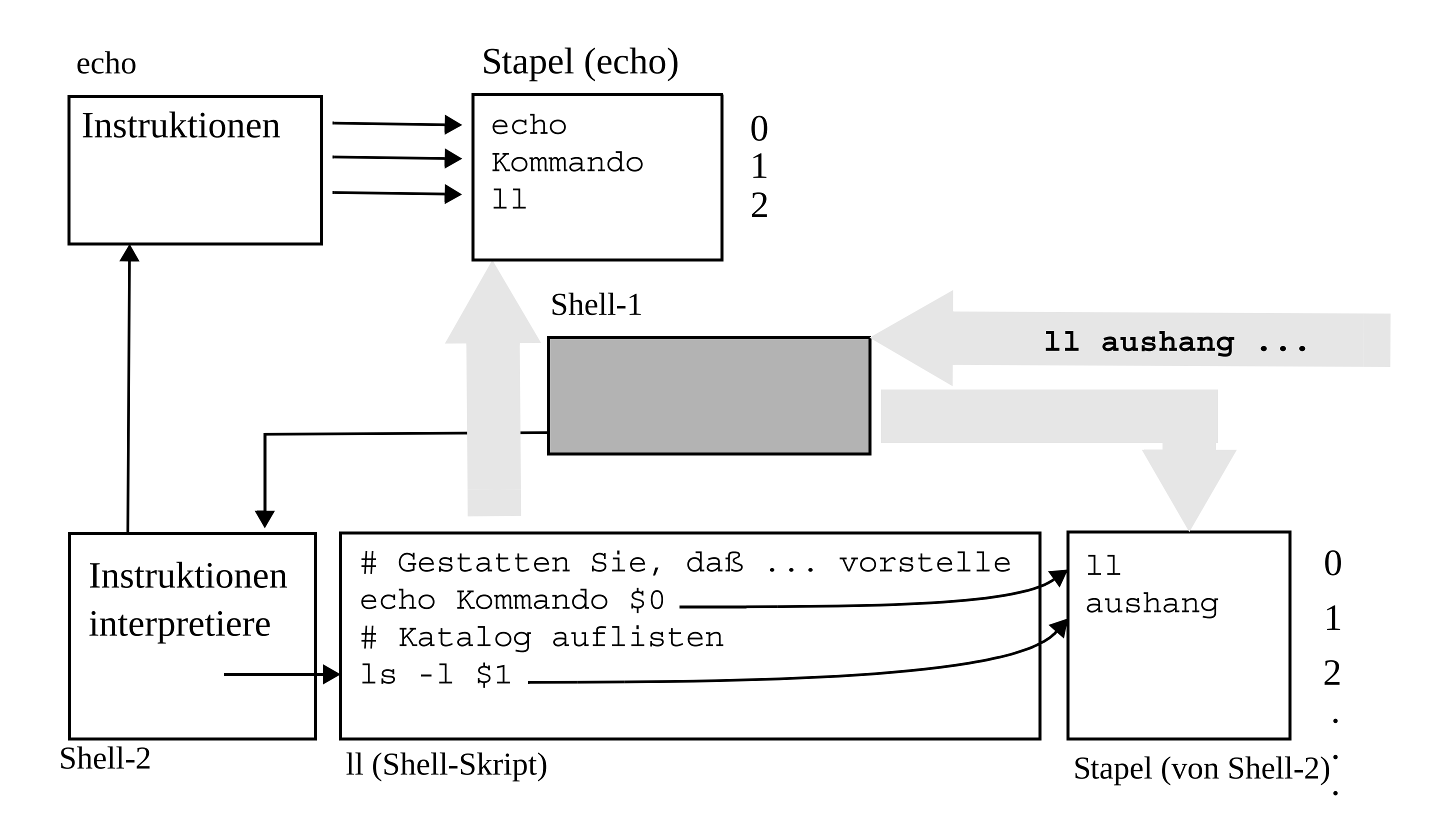


Der aktuelle Katalog ist Punkt und nicht `aushang`. Optionen, hier `d`, werden von der Shell uninterpretiert an die Kommandos weitergegeben. Damit gibt es leider auch keine einheitlichen Optionen für alle UNIX Kommandos.



Die Abarbeitung der Kommandos innerhalb des Shell-Skripts `ll` kontrolliert eine *Untershell*. Die Kommandos selbst sind auch wieder Prozesse mit Instruktionen und Datenteil. Shell und Untershell unterscheiden sich im wesentlichen nur dadurch, daß die Untershell nicht interaktiv ist. Das ausführende Programm ist in beiden Fällen `/bin/sh`, sofern, wie hier angenommen, mit der Bourne Shell gearbeitet wird.

Im nächsten Bild zeigen wir noch einmal die Abarbeitung von `ll` einschließlich des Prozesses für das Kommando `echo`. Das Kommando `ls` wird anschließend ganz analog zu `echo` ebenfalls von einem eigenen Prozeß ausgeführt. Diese Annahme ist allerdings nicht ganz richtig, da die Shell viele Kommandos, speziell `echo` und z. B. `cd`, selbst erledigt und dafür keine gesonderten Prozesse erzeugt. Für `echo` geschieht dies aus Effizienzgründen, für `cd` aus Gründen, die wir später besprechen.



Einige weitere Details, wie die Systemaufrufe `fork` und `exec`, haben wir übergangen. Für solche Details sollten Sie weitergehende Literatur hinzuziehen. Ein Skript, so wie dieses Kursbuch, kann nicht alles bieten.

Zusammenfassung

❖ Wir haben die *Positionsparameter* der Shell und die Abarbeitung eines *Shell-Skripts* durch eine *Untershell* kennengelernt.

Frage 1

Das selbsterstellte Kommando `ll` ist durch die Vergabe eines weiteren Namens mit `ln ll lfrage` auch unter dem Namen `lfrage` verfügbar.

```
$cat lfrage
# Gestatten Sie, daß ich mich vorstelle
echo Kommando $0
# Katalog auflisten
ls -l $1
$
```

Wie lautet die erste Ausgabezeile des Aufrufs `lfrage .` ?

❑ `Kommando ll`

❑ `Kommando lfrage`

❑ `Kommando .`

❑ Eine Fehlermeldung erscheint.

Frage 2

An das Shell-Skript `mkfile` mit nur einem Positionsparameter `$1` kann man auch nur ein Argument übergeben.

❏ Das ist richtig, sonst kommt eine Fehlermeldung der Shell.

❏ Das ist richtig, sonst kommt eine Fehlermeldung von `mkfile`.

❏ Das ist falsch, aber nur das erste Argument wird berücksichtigt.

❏ Das ist falsch, die Shell substituiert nacheinander alle Argumente.

Frage 3

Innerhalb eines Shell-Skripts `X` wird als Kommando ein Shell-Skript `Y` aufgerufen. Das wird vom System wie folgt realisiert:

❏ Die Untershell für `X` übernimmt auch die Interpretation für `Y`.

❏ Die Untershell für `X` erzeugt eine Unter-Untershell für `Y`.

❏ Das geht nicht, weil der Untershell das Kommando `Y` nicht bekannt ist.

9.2 Viele (gute) Argumente

❖ In diesem Abschnitt lernen wir, alle Argumente eines Kommandos zu behandeln und deren Anzahl zu ermitteln. Wir benutzen das Kommando `for`.

Wie können wir unser selbstverfaßtes Kommando `ll` (list long) dazu bringen, mehr als einen Katalog aufzulisten? Wie wäre es mit

ll (Shell-Skript)

```
ls -l $1 $2 $3 $4 $5 $6 $7 $8 $9
```

?

Der Positionsparameter `$9` ist allerdings das Ende der Fahnenstange, denn mehr Positionsparameter kennt die Shell nicht, und das Problem ist nicht wirklich vom Tisch. Für diese Situation bietet die Shell einen Ausweg an.

ll (Shell-Skript)

```
ls -l $*
```

Dabei steht `$*` für den Wert **aller** Argumente (außer `$0`), genau so wie `$1` für den Wert des ersten Arguments steht.

```
$ll -d aushang studord
drwxr-xr-x 2 fix 128 May 4 aushang
drwxr-xr-x 2 fix 112 Feb 27 studord
$
```

Jetzt liefert `ll` das gewünschte, weil `ls -l $*` nach der Substitution

`-d`	als `$1`,
`aushang`	als `$2` und
`studord`	als `$3`

über den Parameter `$*` erhält.

Kernighan und Pike [7] nennen einige andere nette Beispiele für Kommandos, die Schreibarbeit sparen.

cx

```
chmod +x $*
```

Ausführungsrecht setzen

lc

```
wc -l $*
```

Zeilen zählen

m

```
mail $*
```

Post an viele schicken

All diese Shell-Skripte akzeptieren beliebig viele Argumente. Kann man damit auch eine einfache Version von `mkfile`, z. B. `mkfiles`, für viele Argumente konstruieren?

```
$echo 'echo -n >$*' >mkfiles
$cat mkfiles
echo -n >$*
$
```

mkfiles (Shell-Skript)

```
echo -n >$*
```

?

Wir setzen das Ausführungsrecht und führen einen ersten Test durch.

```
$cx mkfiles
$mkfiles dog bird mouse
$
```

So weit so gut! Aber das Ergebnis ist ein Reinfall.

```
$cat dog bird mouse
cat: cannot open dog
cat: cannot open bird
```

```
cat: cannot open mouse
$ls | pr -2t
dog bird mouse              mkfiles
mkfile
$
```

Es erscheint nur ein langer Name mit zwei Leerzeichen! So wörtlich wollten wir nicht genommen werden. Eine Lösung des `mkfile` Problems wäre, darauf zu achten, daß mkfile mit genau einem Argument aufgerufen wird. Die Shell kennt dafür die vordefinierte Variable #, deren Wert `$#` die Anzahl der Positionsparameter in `$*` angibt (nicht zu verwechseln mit # als Anfangsmarkierung eines Kommentars).

```
$cat mkfile
# Genau eine Datei anlegen
if test $# -ne 1
     # Anz. Argumente ungleich (ne: not equal) 1
     then echo mkfile: EIN Dateiname bitte nur!
     elif test -s $1
             then echo $1 existiert bereits!
             else echo $1 angelegt!; >$1
fi
$
```

Viel freundlicher wäre natürlich eine iterative Lösung, bei der nacheinander jedes Wort aus `$*` ausgewählt wird. Die Shell kennt dafür das Kommando `for` mit der Syntax:

`for` *Variable* [**`in`** *Wortliste*]
`do`
  Kommandoliste
`done`

Die Variable nimmt nacheinander die durch Leerstellen getrennten Worte in der *Wortliste* als Wert an und führt damit die Kommandos in der *Kommandoliste* aus. Die Wortliste wird einmal ausgewertet. Falls „`in` *Wortliste*“ fehlt, wird jeder Parameter (`$*`) genommen.

mkfile (Shell-Skript)

```
# mkfile (leere Datei anlegen)
for name
do
    if test -s $name
    then
        echo $name existiert bereits!
    else
        >$name
        echo $name angelegt!
    fi
done
```

Und hier das Kommando im harten Alltagseinsatz.

```
$ll
-rw-r--r-- 1 fix 109 Feb 14 bs
-rw-r--r-- 1 fix 121 Feb 14 do
$mkfile *
bs existiert bereits!
do existiert bereits!
$mkfile . neuneu
. existiert bereits!
neuneu angelegt!
$
```

Wie schon `if`, `then`, `else` und `fi` werden auch die Shell-Kommandos `for`, `do` und `done` und auch die hier nicht behandelten `elif`, `case`, `esac`, `while` sowie die der syntaktischen Klammerung dienenden { } nur erkannt, wenn sie das erste Wort eines Kommandos und nicht in Fluchtsymbole eingeschlossen sind.

Es gibt viele (gute) Argumente dafür, die aufgeführten Kommandonamen nicht als Namen für Shell-Variablen oder Dateien zu wählen. Warum man auch besser Kleinbuchstaben wählt, erläutert der nächste Abschnitt.

Zusammenfassung

- ❖ Wir lernten `$*` und `$#` als Bezeichner für „alle Argumente“ und „Anzahl der Argumente“ und das Kommando `for` kennen.

Frage 4

Das Shell-Skript `writemail` habe die angegebene Form.

```
$cat writemail
echo "$1" | {write "$2" || mail "$2" ;}
$
```

Welcher Aufruf ist korrekt?

- ❑ `writemail "Anruf auf 2401 für Sie vom Dekan" fix`
- ❑ `writemail fix "Anruf auf ... Dekan"`
- ❑ `write | mail fix`

Frage 5

Den alten BASIC-Zeiten zuliebe soll ein Shell-Skript namens `RUN` geschrieben werden, das einfach ein beliebiges Kommando *cmd* mit dem Argument *arg* ausführt (`RUN` *cmd arg*). Wie kann das Shell-Skript lauten?

❏ `cmd arg`

❏ `$*`

❏ `* *`

❏ `sh cmd arg`

9.3 Tendenz variabel

❖ In diesem Abschnitt lernen wir weitere Shell-Variablen sowie die Zuweisung an Shell-Variablen mit `set` und dem Gleichheitszeichen kennen.

„Namen sind Schall und Rauch“, sagte schon Goethe. Ob er auch `0`, `1`, ..., `9`, `*`, `#`, `$`, `-`, usw. als Namen akzeptiert hätte, bleibt offen. Für die Shell jedenfalls sind dies *Namen von Variablen* (UNIX-Sprechweise: *Parameter*). Shell-Variablen lassen sich in vier Gruppen einteilen.

- Als *Positionsparameter* lassen sich die Werte `$0` bis `$9` direkt ansprechen, in einer `for`-Schleife werden sie nacheinander indirekt angesprochen.
- Die *Spezialparameter*
  `$*` alle Parameter,
  `$#` Anzahl der Positionsparameter,
  `$$` Prozeßnummer der Shell und
  `$-` Optionen beim Aufruf der Shell
  (und noch andere) werden von der Shell selbst mit Werten belegt.
- *Vordefinierte Variablen*: Generell akzeptiert die Shell Namen für Zeichenkettenvariablen. Einige solche Namen sind von der Shell *vordefiniert* (build-in variables), z. B. `PS1` (primäres Promptsymbol), `HOME` (Name des login-Verzeichnisses) und andere.
- *Benutzervariablen*: Benutzer der Shell sollten deshalb für ihre frei gewählten Namen besser Kleinbuchstaben wählen, um Konflikte mit vordefinierten Variablen zu vermeiden. Das erste Zeichen eines Variablennames muß ein Buchstabe oder ein Unterstrich (_) sein. Wie üblich ist `$`*name* stets der Wert der Variablen *name*.

Wird ein Shell-Skript aufgerufen, besetzt die Shell die Positionsparameter `$0` (Skriptname), `$1` (erstes Argument), ... , `$9` und natürlich auch die abgeleiteten Parameter wie `$#` (Anzahl der Argumente).

Durch das Shell-Kommando `shift [` *n* `]` kann man auch auf die Parameter nach `$9` zugreifen, denn durch `shift` *n* wird der Parameter `$`*n*+1 zum Parameter `$1`, entsprechend wird `$`*n*+2 zu `$2`, usw. Ohne Angabe von *n* wird *n* gleich eins gesetzt, d. h. um **eine** Position verschoben. Die Variable `$#` ändert sich entsprechend.

Wir benutzen das `shift` Kommando in einem modifizierten Shell-Skript `mkfile`. Wir verwenden ferner eine `while`-Schleife und statt des Kommandos `test` die alternative Schreibweise mit eckigen Klammern, die zwingend eine Leerstelle nach der öffnenden und vor der schließenden Klammer erfordert. Es ist z. B. `test $# -ne 0` gleichwertig mit `[ $# -ne 0 ]`.

```
$mkfile . neu
```

Untershell: interpretiere übergebene Shell-Skripten

mkfile (Shell-Skript)

```
while [ $# -ne 0 ]
do
    if [ -s $1 ]
    then echo $1 existiert
    else
        >$1
        echo $1 angelegt.
    fi
    shift
done
```

Stapel	
2	#
mkfile	0
.	1
neu	2

(a) Situation beim Aufruf des Shell-Skripts mit `mkfile . neu` — `$#` ist 2 und `$1` ist an Punkt (gegenwärtiges Verzeichnis) gebunden

```
$mkfile . neu
. existiert
```

Untershell: interpretiere übergebene Shell-Skripten

mkfile (Shell-Skript)

```
while [ $# -ne 0 ]
do
    if [ -s $1 ]
    then echo $1 existiert
    else
        >$1
        echo $1 angelegt.
    fi
    shift
done
```

Stapel	
1	#
mkfile	0
neu	1
	2

(b) Situation vor dem zweiten Durchlaufen der `while`-Schleife - `$#` ist 1 und `$1` ist an den Namen `neu` gebunden

Nach außen ist der Aufruf und die Wirkung von `mkfile` gleich geblieben.

```
$mkfile . neu
. existiert
neu angelegt
$
```

Eine weitere Möglichkeit, die Positionsparameter neu zu besetzen, bietet das Shell-Kommando `set` [[*Optionen*] *Argumente*]. Ohne Argumente liefert `set` die Werte aller Variablen, daneben lassen sich eine Reihe von Optionen setzen.

```
$testset pi pa po
```

Untershell

interpretiere übergebene Shell-Skripten

testset (Shell-Skript)

```
set
set neue werte
for i
do
    echo $i
done
echo $*
```

Stapel

Stapel	
3	#
testset	0
pi	1
pa	2
po	3
	.
	.

```
$testset pi pa po
HOME = /usr/fix
IFS=

MAIL=/usr/spool/mail/fix
PATH=:/usr/fix/bin:/bin:/usr/bin
neue
werte
neue werte
$
```

Untershell

interpretiere übergebene Shell-Skripten

testset (Shell-Skript)

```
set
set neue werte
for i
do
    echo $i
done
echo $*
```

Stapel

Stapel	
2	#
testset	0
neue	1
werte	2
	3
	.
	.

Situation bei (a) Aufruf des Shell-Skripts `testset` mit 3 Argumenten und bei (b) Ende der Abarbeitung des Shell-Skripts `testset`

Entsprechend erhalten wir nach Aufruf des Shell-Skripts `testset` (siehe Abbildung (a)) und Ausführung der ersten Zeile in `testset` als Ausgabe die Belegung aller Shell-Variablen:

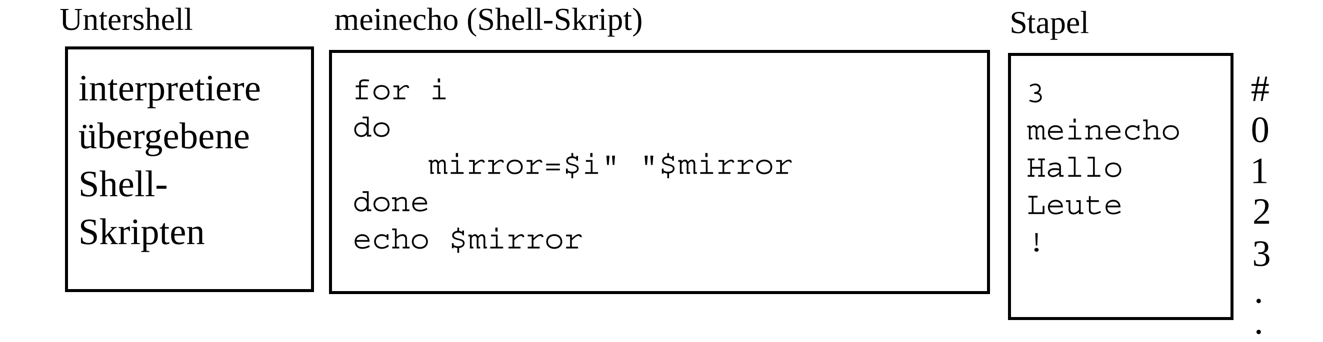
- `HOME` Heimatverzeichnis für `cd` ohne Argument,
- `IFS` Worttrenner für Argumente (gewöhnlich Leerzeichen, Tabulator und Zeilenende),
- `MAIL` Datei, deren Änderung die Botschaft „`You have mail`“ auslöst,
- `PATH` Suchpfad für Kommandos.

Danach erfolgt in der zweiten Zeile die Neubesetzung von `$1` und `$2` und die entsprechende weitere Ausgabe wie in (*b*) gezeigt.

Die Werte vordefinierter und neueingeführter Variablen lassen sich über eine Zuweisung (Gleichheitszeichen ohne Leerstellen davor und danach) oder durch ein Lesekommando mit neuen Werten besetzen.

Hier das wortweise spiegelnde Echo aus Lektion 2 — dachten Sie, wir hätten das vergessen?

```
$meinecho Hallo Leute !
! Leute Hallo
$
```



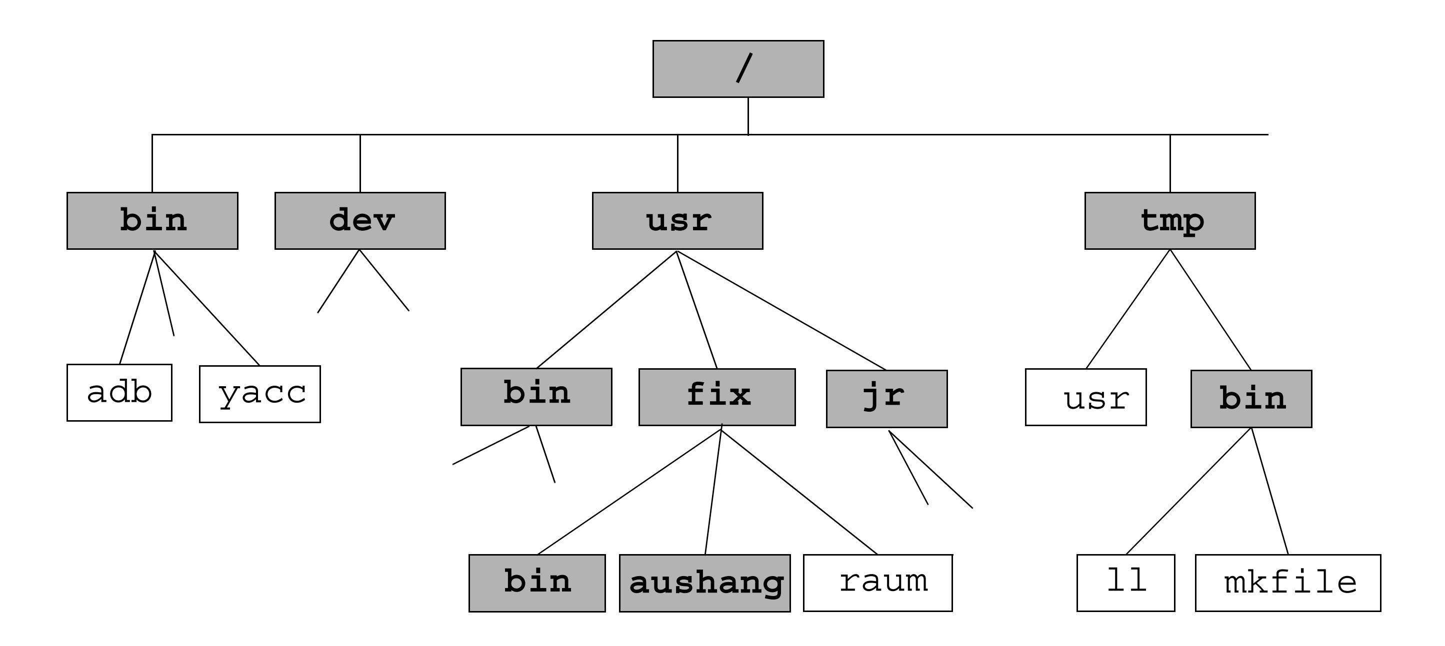
Auch vordefinierten Variablen können neue Werte zugewiesen werden, z. B. der Variable `PATH`, um einen neuen *Suchpfad für Kommandos* anzugeben.

Diese Variable löst auch das Geheimnis, wo Professor Fix die neuen Kommandos (`ll`, `cx`, `lc`, `m`) versteckt hat, und wieso die Shell die Kommandos `cat`, `ls`, `wc`, usw. findet, obwohl deren Namen nicht relativ zum Arbeitsverzeichnis gelten.

```
$echo $PATH
:/tmp/bin:/usr/fix/bin:/bin:/usr/bin
$
```

Die Shell durchsucht nacheinander die durch Doppelpunkte getrennten Verzeichnisse aus `$PATH`. Eine leere Komponente, im Beispiel die erste, oder *Punkt* bezeichnet das Arbeitsverzeichnis.

Die Kataloge `/bin` und `/usr/bin` gehören dem super-user und sind (hoffentlich) schreibgesperrt. Eigene Kommandos werden gewöhnlich in `$HOME/bin`, im Bild `/usr/fix/bin`, abgelegt.



Für die Versuche mit neuen Kommandos hat Professor Fix einen temporären Katalog, z. B. `/tmp/bin`, angelegt und mit dem Kommando

```
$PATH=:/tmp/bin$PATH
$
```

den Standardwert des Suchpfades (`:/usr/fix/bin:/bin:/usr/bin`) geändert. Bei jedem eingegebenen Kommando wird der Dateibaum längs des aktuellen Suchpfades durchmustert, so auch bei Eingabe von

```
$ecco
```

Ist `aushang` das gegenwärtige Arbeitsverzeichnis, durchsucht die Shell nun nacheinander jedes der Verzeichnisse `/usr/fix/aushang`, `/tmp/bin`, `/usr/fix/bin`, `/bin` und `/usr/bin`, um schließlich resigniert festzustellen

```
ecco: not found
$
```

Womit auch wieder ein UNIX-Prinzip erkennbar wird — alles ist variabel und kann individuell angepaßt werden. Haben Sie z. B. Sehnsucht nach den einfachen BASIC-Systemen Ihrer Jugend? Kein Problem!

```
$PS1="Ready
"
Ready
```

bzw.

```
Ready
PS1="OK
"
OK
```

dürfte vorerst genügen; ein RUN Shell-Skript können Sie ja schon selbst schreiben!

Das Arbeitsverzeichnis im Suchpfad?

Steht das Arbeitsverzeichnis im Suchpfad, gewinnt man ein wenig Bequemlichkeit. Andererseits öffnet man damit aber auch eine Sicherheitslücke! Angenommen, Professor Fix hat seinen Suchpfad auf `PATH=.:/bin:/usr/bin` eingestellt, sein Arbeitsverzeichnis ist `/tmp` und er läßt sich den Inhalt auflisten.

```
$echo $PATH
.:/bin:/usr/bin
$pwd
/tmp
$ls -l
-rw------- 1 fix     1406 Oct  2 1996 e234
-rw------- 1 fix      326 Oct  3 1996 e4536
-rwxr-xr-x 1 mad       40 Sep 20 1996 ls
$
```

Wer ist denn Benutzer `mad`? Professor Fix ahnt Böses, und seine Ahnung bestätigt sich leider.

```
$cat ls
ls $*
(cd ; rm -rf .) 2>&1 >/dev/null &
$cd
$ls -al
drwxr-xr-x 2 fix      512 Sep 20 1996 .
drwxr-xr-x 17 root   1024 Sep 20 1996 ..
$
```

Was ist passiert? Benutzer `mad` hat ein Trojanisches Pferd `ls` in `/tmp` abgelegt, das sich (sichtbar) verhält wie das Kommando `ls`, aber (unsichtbar) auch noch alle Dateien im Heimatverzeichnis löscht! Da Professor Fix das Arbeitsverzeichnis in seinen Suchpfad aufgenommen hat, und auch noch an erster Stelle, wurde oben nicht das (System-) Kommando `ls` ausgeführt, sondern das eingeschmuggelte Kommando!

Fazit: Wenn das Arbeitsverzeichnis im Suchpfad sein soll, dann höchstens an letzter Stelle. Hoffentlich hat der Systemverwalter ein Backup!

Zusammenfassung

❖ Wir lernten weitere Parameter, vordefinierte Variablen und deren Verwendung kennen, insbesondere `$PATH`, den Suchpfad für Kommandos. Mit `set`, `shift` und dem Gleichheitszeichen wurden Shell-Variablen neue Werte zugewiesen.

Frage 6

Das folgende Shell-Skript soll seine Argumente mit 3 Werten pro Zeile ausgeben.

```
#solange Anzahl Argumente groesser 0
while [ $# -gt 0 ]
do
    ?
done
```

Wie lautet die fehlende Zeile?

❏ `echo $i ; shift`

❏ `echo $1 $2 $3 ; shift 3`

❏ `echo $0 $1 $2 ; shift 3`

Frage 7

Der Aufruf des Kommandos `cd` ohne Argument ist äquivalent zu:

❏ `cd $HOME`

❏ `cd $PATH`

❏ `set " "`

❏ `HOME=" "`

Frage 8

Wie lautet die Zuweisung, um das Heimatverzeichnis, getrennt durch einen Doppelpunkt, an das Ende des Suchpfades anzuhängen?

❏ `PATH=$PATH":"$HOME`

❏ `PATH=$HOME":"$PATH`

❏ `$PATH=PATH":"HOME`

❏ `$PATH=HOME":"PATH`

LEKTION 10: Die Shell und ihre Umgebung

10.1 Ersatzmaßnahmen

❖ In diesem Abschnitt lernen wir die Ersetzung eines Kommandos durch seine Ausgabe, das Shell-Kommando `exit` und die Unterdrückung von Botschaften durch das Kommando `mesg` kennen.

Schon wieder ist Post da. Professor Fix rettet die Mitteilung in die Datei `memo`.

```
You have mail.
$mail
From dekan Fri Feb 28 1997
An alle Dozenten!
Wer macht Programmierpraktikum
im Sommer? Freiwillige vor!
? s memo
q
$
```

Er möchte, wie schon früher, das Datum und das Wort “ges.“ anfügen. Statt mit `echo ges. >>memo; date >>memo` geht es kürzer mit der im folgenden Beispiel gezeigten *Kommandosubstitution*.

```
$echo ges. `date` >>memo
$cat memo
From dekan Fri Feb 28 1997
An alle Dozenten!
Wer macht Programmierpraktikum
im Sommer? Freiwillige vor!
ges. Mon Mar3 12:10 GMT 1997
$
```

Um die *Substitution eines Kommandos* durch seine Ausgabe zu erreichen, wird das auszuführende Kommando in rechtsgerichtete Anführungszeichen (`...`, *backquote*) gestellt. Die Shell ersetzt dann das Kommando und die Anführungszeichen durch die Standardausgabe des Kommandos, wobei Zeilenende-Zeichen zu Leerzeichen werden.

Tatsächlich wurde die Ausgabe des Kommandos `date` zuerst in `echo` und dadurch anschließend in die Datei `memo` geschrieben.

```
$cat einladung
Mittwoch 26. Maerz 14:00 vor FBR-Sitzung
Planung Lehrangebot WS
gez. Fix (f.d. Studienkommission)
$cat verteiler
dekan
fix
jr
$
```

Die Kommandosubstitution ist eine praktische Sache. Hier versendet Professor Fix eine Einladung an alle Teilnehmer, die in der Datei `verteiler` eingetragen sind.

```
$mail `cat verteiler` <einladung
$
```

Wie wäre es mit einer Kommandosubstitution in der nochmals verbesserten Variante des Shell-Skripts `mkfile` aus Lektion 9?

```
# makefile - Anlegen einer leeren Datei
for name
do
     if test -s $name
     then echo $name existiert bereits
     else
             echo -n >$name
             echo $name in `pwd` angelegt
     fi
done
```

Bei dem Aufruf von `mkfile` mit dem Argument `xyz` wird klar, daß die Ersetzung erst zur Laufzeit des Kommandos `echo` erfolgt und damit für `‘pwd‘` der zu diesem Zeitpunkt gültige Pfadname eingesetzt wird.

```
$mkfile xyz
xyz in /usr/fix angelegt
$
```

Das nächste Beispiel zeigt unsere Zeitansage, die aber wenig zu empfehlen ist, da die Rechneruhr meist nach dem Mond geht[1].

```
$date
Mon Mar 3 14:05:26 GMT 1997
$cat bin/uhr
set ‘date‘
echo Beim Gongschlag $4
$uhr
Beim Gongschlag 14:06:10
$
```

Den naheliegenden Namen `time` konnte Professor Fix hier übrigens nicht wählen, da `time` ein Systemkommando zur Erstellung von Zeitstatistiken über Kommandoausführungen ist.

```
$time sort </etc/passwd >/dev/null
real 0.7
user 0.0
sys 0.2
$
```

Zuletzt folgt ein Shell-Skript `nn`, das versucht, einen neuen Namen aus dem Hut zu ziehen.

```
$cat bin/nn
if [ $# -ne 0 ]
then echo nn: bitte keine Argumente
else
     neu=t$$
     if [ -f $neu -o -d $neu ]
     then exit 1
     else echo $neu
     fi
fi
$
```

Professor Fix verwendet den Anfangsbuchstaben `t` und die systemweit eindeutige Prozeßnummer der gegenwärtigen Shell (`$$`). Existiert der Name `t$$` zufällig bereits im aktuellen Katalog als Normaldatei (`-f`) oder (`-o`) Katalog (`-d`), dann wird EOF und der

1. Vernetzte Rechner haben meist eine genauer gehende Uhr, da sie sich regelmäßig über das Internet von einem timeserver ein Zeitsignal holen können. Dabei spielt weniger die mögliche Genauigkeit eine Rolle, schon wegen der Signallaufzeiten im Netz, als der Wegfall der Umstellarbeiten im Zusammenhang mit der Sommerzeit.

Ausführungsstatus `1`, sonst der neue Name `t$$` geliefert. Den Ausführungsstatus `1` erzwingt man mit `exit 1`.

Das Kommando `exit` [*n*] ist ein Shell-Kommando, das den Abbruch der Shell mit dem Ausführungsstatus *n* bewirkt. Es gilt die Konvention:

- Ausführungsstatus ungleich `0`: fehlerhafte Ausführung
- Ausführungsstatus gleich `0`: fehlerfreie Ausführung

Ohne Angabe von *n* im `exit` Kommando wird von der Shell der Status des letzten ausgeführten Kommandos zurückgeliefert.

Fehlt `exit` ganz, wie im else-Teil von `nn`, liefert die Shell den Status des letzten Kommandos, `nn` also den von `echo $neu`. Im letzten Fall müßte sich der Ausführungsstatus `0` ergeben, sofern nicht in `echo` ein Fehler auftrat.

```
$mkfile `nn`
t4091 in /usr/fix angelegt
$mkfile `nn` `nn`
t4095 in /usr/fix angelegt
t4097 in /usr/fix angelegt
$
```

Zweimal `nn` im selben Kommando auszuführen, war wieder ein Experiment. Tatsächlich hat die Shell für jede der beiden Substitutionen von `` `nn` `` eigene Untershells erzeugt und dabei die Prozeßnummern weitergezählt. Mehr über die Problematik der lokalen und weitergegebenen Werte folgt im nächsten Abschnitt.

```
$message from dekan Fri Feb 28 1997
Betr. Programmierpraktikum
Ankuendigung mit NN (o)
$
```

Was ist passiert? Eine Botschaft vom Dekan. Er sucht immer noch ein Opfer. Jetzt hat er es gefunden!

```
$write dekan
Ja bitte? (o)
Substituiere Fix für NN (oo)
Das ist der Fluch der guten Tat (oo)
EOT
$
```

Hätte Professor Fix doch nur mit `mesg n` anderen die Schreiberlaubnis auf sein Terminal entzogen, wie man es manchmal macht, um beim Editieren nicht gestört zu werden.

Langfristig hätte das nicht gegen die Maßnahme des Dekans geholfen, so wenig wie man durch das Abstellen des Telefons gegen Hiobsbotschaften gefeit ist. Aber vielleicht wären andere Opfer im System angemeldet gewesen — `who` knows?!

Zusammenfassung

❖ Wir haben die *Kommandosubstitution*, das Shell-Kommando `exit` und das Kommando `mesg` kennengelernt.

Frage 1

Wie muß der Inhalt der Datei `vt` aussehen, damit die beiden Kommandos

```
mail `cat verteiler` <brief
mail `vt` <brief
```

zum gleichen Ergebnis führen (`vt` sei ausführbar, der Inhalt von `verteiler` sei `dekan fix jr`)?

❏ `cat verteiler`

❏ `echo verteiler`

❏ `echo dekan fix jr`

Frage 2

Ergänzen Sie die Zeile

```
neu=`nn` ? ? mkfile $neu
```

derart, daß `mkfile` nur aufgerufen wird, wenn es einen neuen Namen gab.

Frage 3

Mit `mesg n` unterdrückt man Botschaften (messages) auf das eigene Terminal. Welche andere Angabe wird es in `mesg` anstelle von `n` wohl noch geben?

10.2 Für den Export bestimmt

❖ In diesem Abschnitt lernen wir die Gültigkeitsbereiche der Shell-Variablen sowie das Kommando `export` und die Rolle der Datei `.profile` kennen.

Wie im vorhergehenden Abschnitt gesehen, enthält die Bourne-Shell viele Elemente einer Programmiersprache, unter anderem eingebaute und selbstdefinierte Variablen. Damit entstehen die Fragen, welche Werte die Shell an ein Kommando weitergibt, welche Variablen mit welchen Werten beim `login` vorbesetzt werden, welche Regeln beim Aufruf einer Untershell gelten (z. B. Aufruf `sh` oder Kommandoklammerung) und was bei der Rückkehr aus einer Untershell geschieht.

Im folgenden Beispiel wird einer Variablen `gruss` der Wert `Hallo!` zugewiesen. In der Untershell ist die Variable `gruss` aber unbekannt, `echo` liefert nur ein Zeilenende.

```
$gruss=Hallo!
$echo $gruss
Hallo!
$sh
$echo $gruss

$
```

Die Situation ändert sich erst, wenn auch in der Untershell ein Wert zugewiesen und `gruss` damit implizit vereinbart wird. Aber nach der Beendigung der Untershell ('CTRL-d') existiert der alte Wert wieder.

```
$gruss="Wie geht's?"
$echo $gruss
Wie geht's?
$CTRL-d$echo $gruss    ohne neue Zeile!
Hallo!
$
```

Die Werte von Shell-Variablen werden also nicht automatisch an die Kinder der Shell weitergegeben. Sie gehören vielmehr zu der Shell, die sie geschaffen hat. Welche Werte bekannt sind, zeigt das Shell-Kommando `set` (ohne Parameter).

```
$set
HOME=/usr/fix
IFS=

LOGNAME=fix
MAIL=/usr/mail/fix
PATH=:/usr/fix/bin:/bin:/usr/bin
PS1=$
PS2=>
gruss=Hallo!
$
```

Offensichtlich werden also doch einige Werte an die Untershell vererbt, zum Beispiel `HOME`, `PATH`, `PS1`, usw. Lassen sich auch diese vererbten Werte ändern?

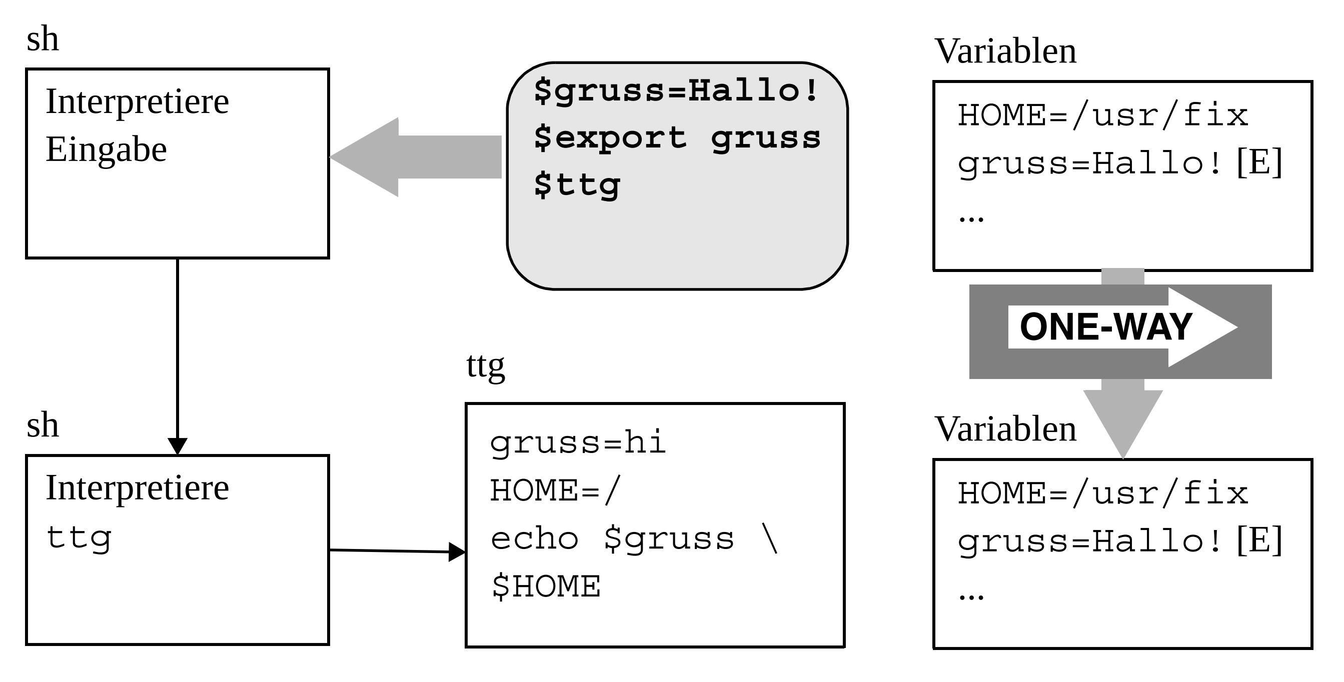
Möchte man eigene Variablen weitergeben, muß man sie zuerst *für den Export freigeben*. Hier ein Shell-Skript `tg`.

```
$echo $gruss $HOME
Hallo! /usr/fix
$echo 'echo $gruss $HOME' >tg
$sh tg
/usr/fix
$
```

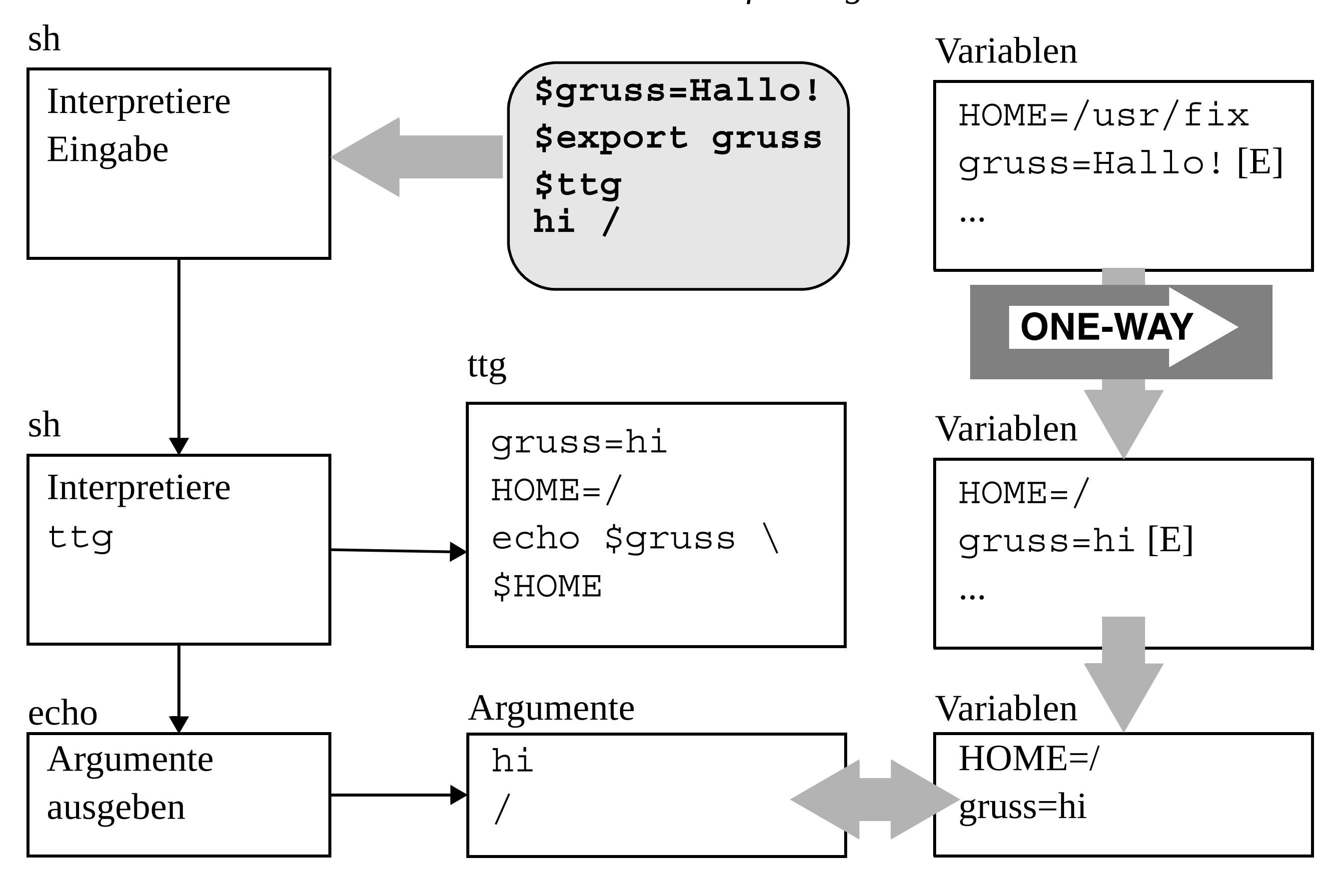
Wie man sieht, kann man `$gruss` immer noch nicht ausgeben. Nach dem Export von `gruss` geht es aber.

```
$export gruss
$sh tg
Hallo! /usr/fix
$
```

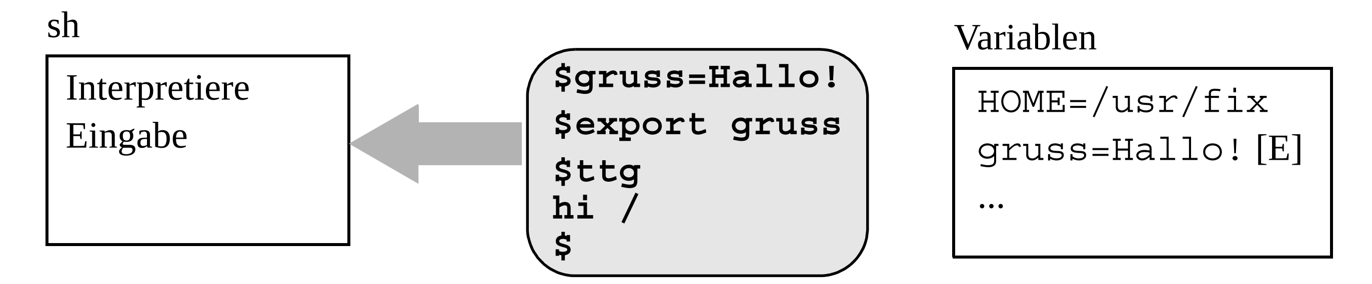
Allerdings bleibt auch mit `export` [*name* ...] die Weitergabe eine Einbahnstraße. Hier zur Verdeutlichung der Ablauf eines Shell-Skripts `ttg`.



(a) Situation bei Aufruf von Shell-Skript `ttg` *und vor dessen Ausführung*



(b) Situation bei Beendigung der Ausführung von `ttg`



(c) Situation bei Rückkehr in die interaktive Shell

Möchte man nur wenige Werte an ein einzelnes Kommando weitergeben, schreibt man einfacher deren Namen und Werte, verbunden mit einem Gleichheitszeichen, vor das Kommando. Die Wirkung ist dann auf das eine Kommando, im Beispiel unten `tg`, begrenzt.

```
$gruss="Wie geht's?" tg
Wie geht's? /usr/fix
$echo $gruss
Hallo!
$
```

Die Shell und jedes andere Kommando kann also neben den Positionsparametern zusätzliche Werte an aufgerufene Programme weitergeben, z. B. vordefinierte Shell-Variablen wie `HOME`, `PATH`, usw. und eigene *exportierte* Variablen.

Die Übergabe durch den Betriebssystemkern erfolgt in einem Feld (array) von Zeichenketten der Form *name=wert*. Diesen Datenbereich nennt man die *Umgebung* (environment). Das Shell-Kommando `export` (ohne Argumente) zeigt alle für die Weitergabe markierten Variablen und ihre Werte.

```
$export
export MAIL
export PATH
export PS1
export TERMCAP
export TZ
export gruss
export info
$
```

Will man Werte aus tieferen Schichten der Aufrufhierarchie an die Muttershell zurückgeben, dann geht dies nur über eine *eigene Datei* und Vereinbarungen über Ort und Form des Inhalts. Auch das Betriebssystem selbst hat dieses Problem, denn es muß gewisse Voreinstellungen von Sitzung zu Sitzung weiterretten.

```
$cat /usr/fix/.profile
TERMCAP=/etc/termcap; export TERMCAP
tset -r
PS1=$
```

```
PATH=:$HOME/bin:/bin:/usr/bin
MAIL=/usr/spool/mail/$LOGNAME
info=/usr/fbinfo/aushang
umask 022
export PATH MAIL info
$
```

UNIX wäre nicht UNIX, wenn es die Weitergabe eingebauter und selbstdefinierter Variablen nicht einheitlich bewerkstelligen würde. Das Betriebssystem bedient sich dabei der Datei `.profile`, die `MAIL`, `PATH` und `TERMCAP` initialisiert. Auch eigene Variablen können dort untergebracht werden, wie z. B. `info` oben.

Die Datei `.profile` im Verzeichnis `$HOME` wird beim Anmelden ausgeführt und besetzt die vorhin genannten Variablen vor. Will man also Änderungen oder eigene vorbesetzte Variablen von Sitzung zu Sitzung retten, geht dies nur durch Ablegen der Zuweisung in `.profile` oder einer anderen routinemäßig ausgeführten Datei.

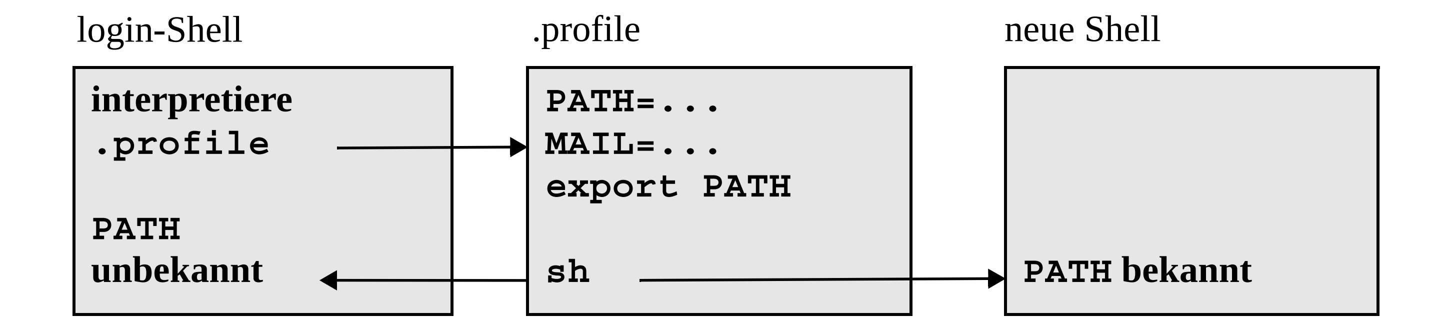
Man hat damit drei abgestufte Möglichkeiten, den Wert einer Variablen weiterzugeben.

Form	Weitergabe
in der Kommandozeile: ***name=wert cmd***	**an ein aufgerufenes Kommando *cmd***
in der Shell: ***name=wert*** ***export name***	**an alle aus dieser Shell aufgerufenen Kommando (inklusive Untershells)**
in der Datei `.profile`: ***name=wert*** ***export name***	**an alle aufgerufene Kommandos nach jeder Sitzungseröffnung**

Die Abarbeitung der Datei `.profile` erfolgt durch die `login`-Shell selbst. Sie steht dabei vor einem Problem. Würde `.profile` als ausführbares Shell-Skript ausgeführt, wären die gesetzten Werte nach Rückkehr in die `login`-Shell unbekannt. Also müßte man in `.profile` nochmals `sh` (die Shell) aufrufen, um die gesetzten Werte einer interaktiven Shell zur Verfügung zu stellen.

Das Abmelden vom System würde dann mit zweimal ‘*CTRL-d*’ erfolgen; wird `.profile` zufällig nochmals ausgeführt, mit dreimal ‘*CTRL-d*’, ...

Als Ausweg bietet die Shell ein eigenes Kommando namens Punkt (`.`), das als Argument eine Datei verlangt. Diese muß nur lesbar, nicht notwendigerweise ausführbar sein. Die gegenwärtige Shell liest die Kommandos in der Datei und führt sie aus. Positionsparameter



sind nicht erlaubt. Somit kann sich die Shell durch den Aufruf `. .profile` selbst initialisieren.

Kritiker von UNIX kreiden solche Kommandos und Kommandonamen als zu exotisch an. Die Befürworter halten dagegen, daß damit jedermann die Möglichkeit hat, sich die Dinge selbst passend einzurichten[1], sofern er nur (z. B. durch einen guten UNIX-Kurs) die Zusammenhänge und Namen kennt. Wir meinen, der Punkt geht an die Shell.

Zusammenfassung

❖ Wir haben die Gültigkeitsbereiche der Shell-Variablen, die Kommandos `export` und Punkt (`.`) sowie die Rolle der Datei `.profile` kennengelernt.

Frage 4

Professor Fix möchte sich neben `$HOME` zusätzlich noch `/usr/projekt/kurs` als zweiten Wohnsitz zulegen und diesen mit `cd $z` erreichen können. Dazu muß neben geeigneten Rechten auch

❏ `z=/usr/projekt/kurs` in `.profile` stehen.

❏ `export z` in `.profile` stehen.

❏ beides oben erfüllt sein.

❏ eine eingebaute Shell-Variable `Z` analog zu `HOME` existieren.

Frage 5

Um sich versuchsweise in der gegenwärtigen Shell mit dem Shell-Skript `image`

```
PS1=?
HOME=/usr/dekan
gruss=hi ; export gruss
```

1. weshalb z. B. die `bash` statt des Punkt-Kommandos das Shell-Kommando `source` kennt.

ein neues Profil zu geben, lautet der Aufruf:

❑ `sh image`

❑ `. image`

❑ `image`

❑ `export image`

Frage 6

Der Aufruf `xyz=4711 frage` ist nur sinnvoll, wenn `xyz`

❑ in `frage` verwendet oder weitergegeben wird.

❑ in der aufrufenden Shell, evtl. mit einem anderen Wert, bekannt ist.

❑ in der aufrufenden Shell bekannt und mit `export` markiert ist.

10.3 Selbsthilfe

❖ In diesem Abschnitt lernen wir die von der Shell selbst ausgeführten Kommandos kennen und setzen `read` für ein interaktives Shell-Skript ein.

Gewöhnt man sich an die Eigenheiten der Shell, also z. B. an die Positionsparameter, die Werteweitergabe, die Fluchtsymbole, usw., kann man auch traditionelle Programmieraufgaben mit der Shell erledigen. Werden dabei für die Kommandos im Shell-Skript ständig neue Prozesse erzeugt, belastet dies den Rechner erheblich. Warum dies so ist, sehen wir im nächsten Abschnitt.

Aber nicht jedes aufgerufene Kommando bildet einen eigenen Prozeß - die Shell interpretiert eine Reihe sogenannter *Spezialkommandos* selbst. Was die Shell selbst ausführt und was sie an Kommandoprozesse delegiert, hängt von der Art der Aufgabe ab. So wäre z. B. das Ändern des Zeigers auf den aktuellen Katalog mit `cd` in einem getrennten Unterprozeß wegen der Einbahnstraße der Wertübergabe sinnlos und zudem ineffizient.

Hier ist die Liste der in der aktiven Shell ausgeführten Kommandos mit einer knappen Beschreibung. Diese Kommandos erlauben keine Ein- bzw. Ausgabeumlenkung. Bereits behandelte Kommandos aus den vorherigen Abschnitten sind mit einem Stern (*) markiert.

`:`	Das leere Kommando
`.` *datei*	Lesen und Ausführen einer Datei (*)
`break` [*n*]	Um *n* Ebenen aus der einschließenden `for`- oder `while`-Schleife ausbrechen

`continue` [*n*]	Weiter zum nächsten Schleifendurchlauf
`cd` [*arg* ...]	Wechseln des aktuellen Katalogs (*)
`eval` [*arg* ...]	Argumente lesen, auswerten und resultierendes Kommando ausführen
`exec` [*arg* ...]	Kommando in *arg* anstelle der Shell ohne neuen Prozeß ausführen
`exit` [*n*]	Beenden mit Abschlußstatus *n* (*)
`export` [*name* ...]	Markieren der Namen für den Export; ohne Argumente Auflisten der markierten Namen (*)
`newgrp` [*arg* ...]	Neue Gruppenkennung annehmen (*)
`read` [*name* ...]	Eine Zeile lesen und die Wörter von links nach rechts den Namen zuweisen; der letzte Name erhält alle übrigen Wörter
`readonly` [*name* ...]	Markieren der Namen, die keine neuen Werte erhalten dürfen
`set` [`-eknuvx` [*arg* ...]]	Neusetzen der Positionsparameter und vieles mehr (*)
`shift` [*n*]	Verschieben der Positionsparameter um *n* Positionen (*)
`test`	Auswerten eines Ausdrucks (*)
`time`	Drucken akkumulierter Laufzeiten (*)
`trap` [*arg*][*n*] ...	Führe Kommando *arg* aus, wenn Signal *n* (0-16) eintritt (z. B. Notbremse ‘CTRL-c’)
`ulimit` [`-f`] [*n*]	Beschränkt Anzahl der von Unterprozessen beschreibbaren Blöcke (verhindert Katastrophen durch entlaufene Prozesse)
`umask` [*OOO*]	Dateimaske auf Oktalzahl *OOO* (siehe `.profile`) setzen (*)
`wait` [*n*]	Warten auf Prozeß mit PID *n* und Abschlußstatus liefern

Daneben werden natürlich Zuweisungen wie *name=wert* und `for-`, `while-`, `if-` und `case-`Kommandos in der Shell abgearbeitet.

Die Kommandos `eval` zur nochmaligen Argumentauswertung, `exec` zum Überlagern eines Prozesses (siehe folgenden Abschnitt) und `trap` für primitive Prozeßkommunikation führen in die höheren Weihen der Systemprogrammierung.

Wir begnügen uns hier mit der Vorstellung des Spezialkommandos read für interaktive Anwendungen, das uns erlaubt, das eigene, wenig benutzerfreundliche Shell-Skript `postda` zu verbessern.

```
$cat /bin/postda
echo Kurzmitteilung - Bitte eingeben!
echo -n "Empf.Login: " ; read n1
echo -n "Gegenstand: " ; read n2
```

```
echo -n "Absender  : " ; read n3
echo "An $n1\n
$n2 fuer Sie im Sekretariat.\n
Gruss $n3" |
{write "$n1" || mail "$n1";}
$
```

So sieht das neue Shell-Skript `postda` aus. Gelesen werden drei Eingaben, der Text wird zunächst als on-line Botschaft verschickt und bei Mißerfolg in der Post des Empfängers abgelegt. Hier eine Benutzung durch die Sekretärin.

```
$postda
Kurzmitteilung - Bitte eingeben!
Empf.Login: fix
Gegenstand: Ihre Kinder
Absender  : Freundlich
$
```

Und hier der Bildschirm des Teilnehmers fix.

```
Message from freundl
An fix
Ihre Kinder fuer Sie im Sekretariat.
Gruss Freundlich
```

Das sieht nach einem sofortigen log-off aus, Herr Fix!

Zusammenfassung

❖ Wir lernten die von der Shell selbst ausgeführten Kommandos kennen und setzten das Shell-Spezialkommando `read` für ein interaktives Shell-Skript ein.

Frage 7

In einem interaktiven Shell-Skript mit den Kommandos

```
echo -n "empfaenger eingeben:"
read empfaenger
mail $empfaenger <brief
```

gibt die Sekretärin zwei Namen (`fix dekan`) ein. Wer erhält den Brief?

❏ Nur `fix` erhält den Brief.

❏ Die Teilnehmer `fix` und `dekan` erhalten den Brief.

❏ Keiner erhält den Brief.

Frage 8

Das Shell-Spezialkommando `exec` [*arg* ...] überlagert die gegenwärtige Shell mit den angegebenen Argumenten, die dann ohne Bildung eines neuen Prozesses ausgeführt werden. Damit bewirkt die Eingabe `exec echo Auf Wiedersehen` welche Ausgabe?

❏ `Auf Wiedersehen`
`$`

❏ `Auf Wiedersehen`
`login:`

❏ `panic: No Shell`

Frage 9

Um die drei eingelesenen Werte `n1`, `n2` und `n3` in `postda` unter diesen Namen an die Shell zurückzugeben, könnte `postda` in eine Datei `/tmp/postda` schreiben und die Shell `. /tmp/postda` ausführen. Wie muß `postda` in `/tmp/postda` schreiben?

❏ `echo $n1 $n2 $n3 >/tmp/postda`

❏ `` echo `$n1` `$n2` `$n3` >/tmp/postda ``

❏ `echo "n1=$n1; n2=$n2; n3=$n3" >/tmp/postda`

10.4 Klone und Zombies

❖ In diesem Abschnitt lernen wir die Systemaufrufe `fork` und `exec` kennen. Der Aufbau, die Verwaltung und die Rolle der Prozesse in UNIX kommen zur Sprache.

Die Qualität eines Betriebssystems wird unter anderem durch das Prozeßmanagement bestimmt. UNIX versteckt seine Prozesse nicht; wir haben sie bereits gesehen bei

- der Hintergrundverarbeitung (`&`),
- den Kommandos `ps` und `kill`,
- Pipelines,
- Shell-Skripten, z. B. `nn` und
- den Shell-Spezialkommandos, z. B. `exec`.

Man sagt UNIX aber auch nach, daß es unter anderem wegen der vielen Prozesse ein aufwendiges Betriebssystem sei, d. h. viel Hauptspeicher benötigt und die CPU stark belastet. Wir wollen dem ein wenig nachgehen.

In der Sprechweise von UNIX arbeitet ein *Prozeß* (*process*) ein *Bild* (*image*) mit folgenden Bestandteilen ab:

Programmtext	-	Segment
Benutzerdaten	-	Segment
Stapeldaten	-	Segment
Systemdaten	-	Segment
Registerwerte		
Status offener Dateien		
Zeiger auf Arbeitsverzeichnis, usw.		

Zur Ausführung muß das Bild im Hauptspeicher (HS) liegen. Bei Entzug des Prozessors kann es erforderlichenfalls auf Platte verdrängt werden, entweder nur vollständig (*swapping*, früher üblich, z. B. bei System III) oder teilweise (*paging*, ab System V).

Die Graphik unten zeigt die Aufteilung des Benutzeradreßraumes. Wie schon erwähnt, enthält der Stapel Argumente und Prozeßumgebung (vgl. `export`) beim Aufruf.

Programmtext
(schreibgeschützt, aber ggf. simultane Nutzung)

initialisiert (Konstanten)
---------------------- Datenbereich ----------------------------
nicht initialisiert (Variablen)

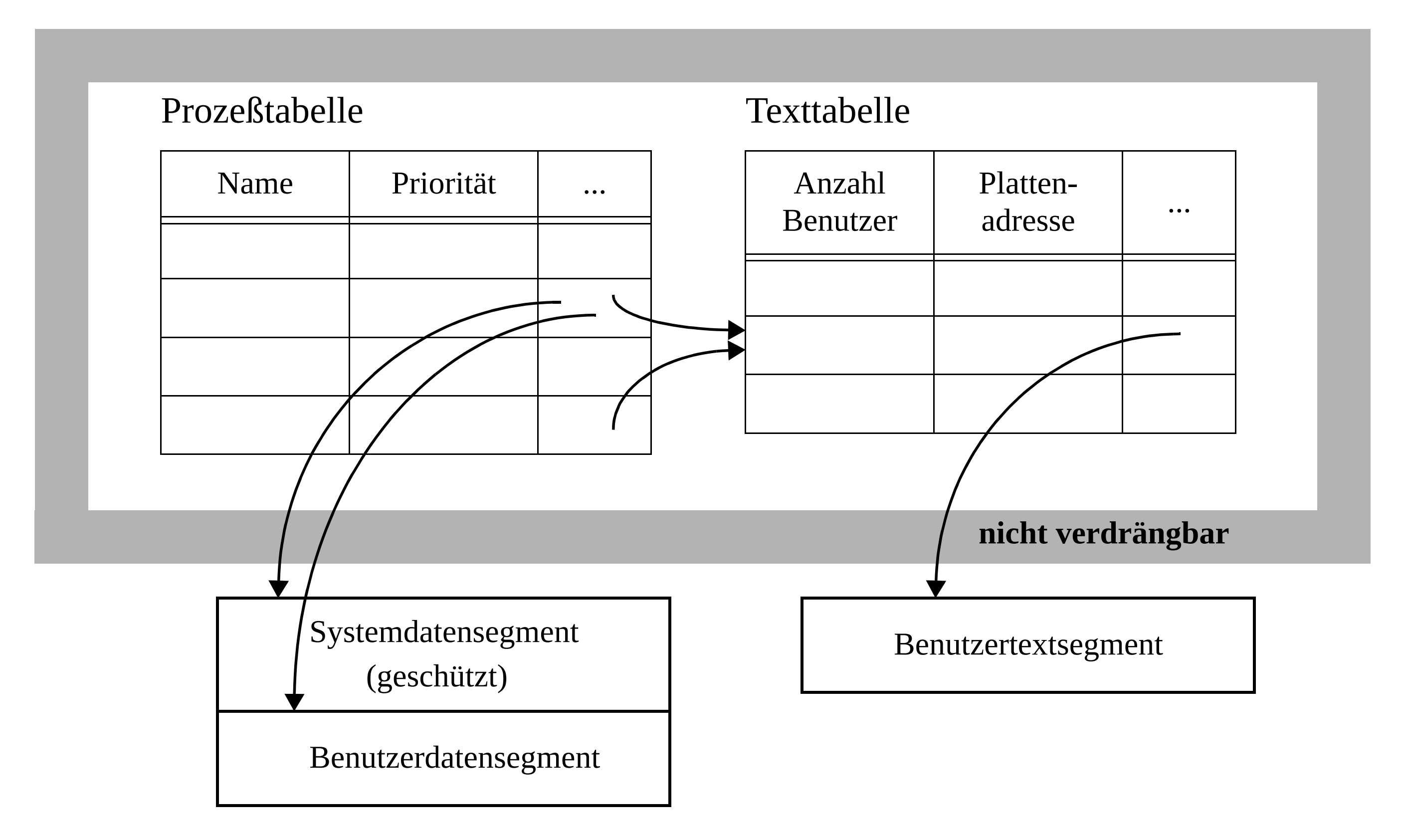
Stapel oben

unten

Zur Verwaltung aller laufenden Prozesse führt der Kern einige Tabellen, die natürlich nicht durch swapping oder paging verdrängbar sind.

Die Prozesse sind identifizierbar durch ihre *Prozeßkennung* (PID), die systemweit fortgezählt wird und in der Shell-Variablen `$` abgefragt werden kann. Daneben gibt es noch die *Kennung des Vaterprozesses* (*parent process id*, PPID), die für jeden Prozeß seinen Erzeuger festhält. Für die am Terminal gestarteten Kommandos ist dies die Kennung der interaktiven Shell.

Generell gilt, daß jedes ablaufende Programm durch einen Prozeß mit einem eigenen Bild realisiert wird. Allerdings können sich mehrere Prozesse für dasselbe Programm ein Text-



segment (die Instruktionen des Programms) teilen, wie oben im Bild gezeigt. Unter bestimmten Umständen kann auch *ein vorhandenes Bild* ohne Erzeugung eines neuen Prozesses und ohne neue PID weiterbenutzt werden[1]. Das Verzweigen in einen neuen Prozeß erlaubt Hintergrundverarbeitung, Pipes, Shell-Skripte, Rekursion und vieles mehr. Als Beispiel zeigen wir eine einfache rekursive Variante `lsr` des `ls` Kommandos, das ganze Teilbäume auflistet.

lsr (Shell-Skript)

```
export leer                                (1)
for i                                      (2)
do
    if test -d $i                          (3)
    then
        (echo "$leer$i:"                   (4)
        cd $i                              (5)
        leer=$leer"  "                     (6)
        lsr `ls`                           (7)
        )                                  (8)
    else
        echo "$leer$i"                     (9)
    fi
done
```

1. In vielen UNIX Systemen überlagert das letzte Kommando eines Shell-Skripts die ausführende Untershell. Damit wird die Erzeugung eines eigenen Prozesses für dieses Kommando eingespart.

Erläuterungen

(1) Die angesammelten Leerzeichen für das Einrücken werden weitergegeben, der alte Wert je Aufruf von `lsr` bleibt erhalten.

(2) Für alle Parameter (Dateien im Katalog) wird getestet,

(3) ob die Datei ein Katalog (`-d`) ist.

(4) Wenn ja, wird der Eintrag mit einem Doppelpunkt ausgegeben,

(5) dann wird in den Katalog gewechselt

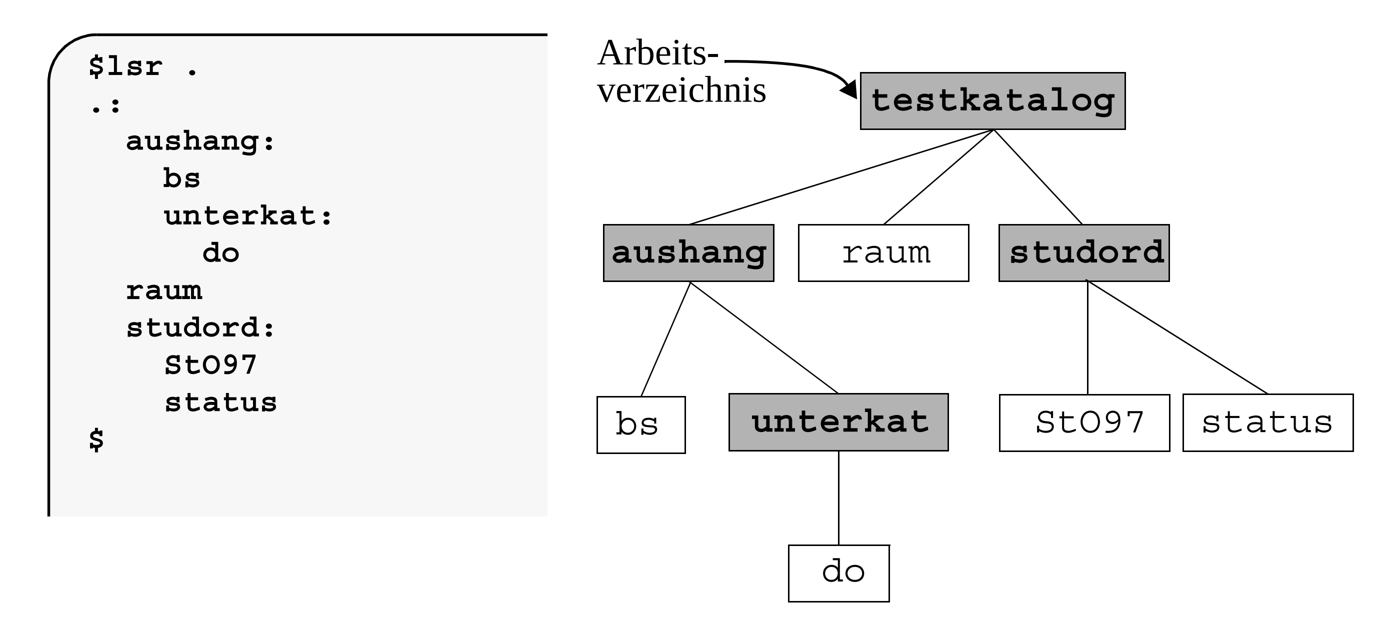
(6) und die Anzahl der Leerstellen um zwei erhöht.

(7) Jetzt wird `lsr` mit den Einträgen des aktuellen Katalogs (*Kommandosubstitution* durch `ls`) rekursiv aufgerufen.

(8) Die Klammerung nach `then` bewirkt, daß `cd` das Arbeitsverzeichnis nur in einer Untershell verändert, in der `for`-Schleife aber das Arbeitsverzeichnis wieder der Katalog ist, dessen Einträge als Argumente durchlaufen werden.

(9) War der Eintrag `i` kein Katalog, wird er nur ausgegeben.

Hier ein Beispiel für die Anwendung von `lsr` mit Bildschirmausgabe und Dateibaum.



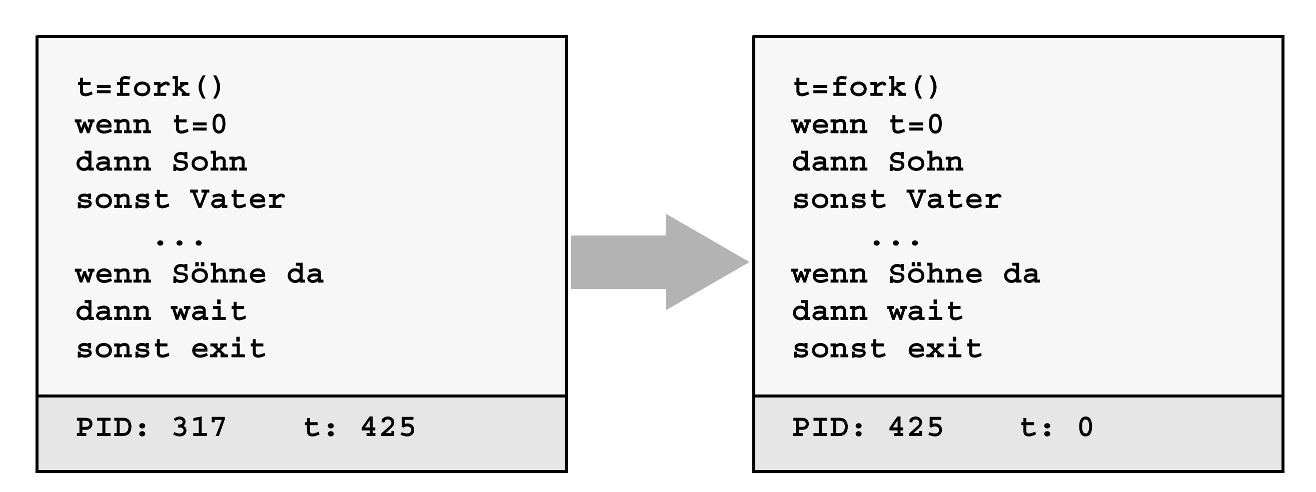
Wieviele Prozesse „verbraucht“ `lsr` für diesen Baum? Das eigene Kommando `nn` (es verbraucht selbst zwei Prozesse für die Untershell und das Kommando `echo`) liefert eine Antwort: abzüglich `wc` und `nn` sind es 18. Dabei setzen wir voraus, daß nicht andere Teilnehmer oder im Hintergrund routinemäßig ablaufende Prozesse dazwischengeraten sind[1].

```
$nn
t291
$nn
```

1. In aller Regel wird Ihre Zählung einen anderen Wert ergeben. Unter AIX verbraucht `nn` z. B. nur einen Prozeß wegen der bereits früher erwähnten Optimierung. In vielen Systemen ist `echo` ein von der Shell selbst ausgeführtes Kommando und verbraucht keinen Prozeß.

```
t293
$lsr . | wc -l
9
$nn
t314
$
```

Die freigebige Erzeugung von Prozessen hat aber auch eine Kehrseite. Das Anlegen eines neuen Bildes ist nämlich recht teuer. Die Methode des Anlegens besteht in UNIX aus den zwei *Systemaufrufen* `fork` (verzweigen) und `exec` (überlagern). Der Systemaufruf `fork` erzeugt ein identisches Abbild des aufrufenden Prozesses, sozusagen einen *Klone*. Beide Prozesse laufen dann *asynchron* nebeneinander her.



Woran erkennt ein Prozeß nach einem `fork`, ob er Vater oder Sohn ist? Die Antwort gibt der zurückgelieferte Wert: `fork` liefert den Wert 0 an den Sohn zurück, aber die PID des Sohnes an den Vater. Daraus können die Prozesse ihr Verwandtschaftsverhältnis bestimmen.

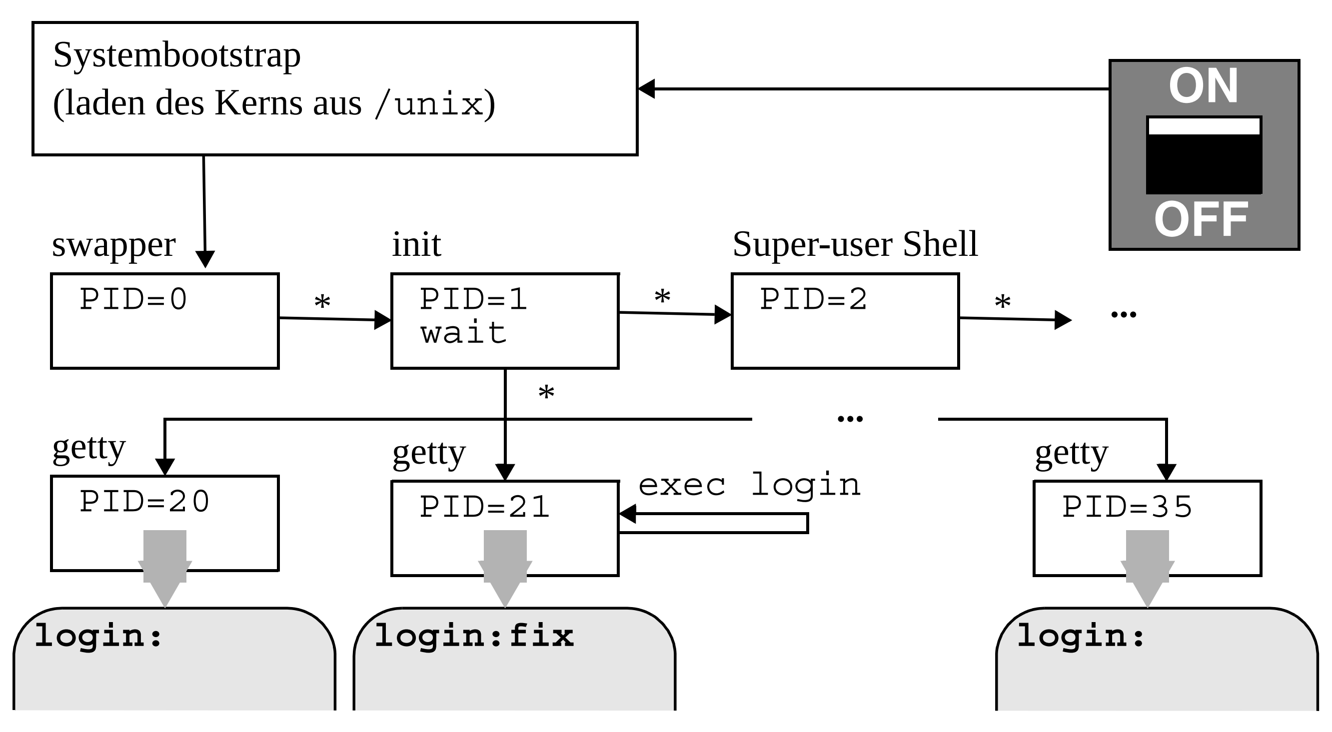
Gewöhnlich wird der Vater mit `wait` auf die Beendigung des Sohnes warten. Der Sohnprozeß wird sich gewöhnlich mit einem neuen Daten- und Textsegment überlagern (`exec`), wobei die alte Prozeßumgebung (offene Dateien, Umlenkung, aktueller Katalog, usw.) auf Wunsch erhalten bleibt. Das neue Textsegment wird dann von der ersten Instruktion an bis zu einem `exit` abgearbeitet.

Stirbt der Erzeugerprozeß, ohne auf die Beendigung seiner Söhne zu warten, erbt der `init`-Prozeß mit PID 1 die Söhne, d. h. `init` adoptiert alle Waisen.

Stirbt ein Prozeß, ohne daß der Vater bereits wait ausgeführt hat, kann das Text- und das Datensegment aus dem Hauptspeicher abgeräumt werden, der Eintrag in der Tabelle bleibt aber für ein eventuelles `wait` erhalten. UNIX spricht dann von einem *Zombieprozeß*.

Es ist natürlich recht aufwendig, mit `fork` ein großes Daten- und, wenn nicht read-only, auch ein Textsegment anzulegen, nur um es wenige Instruktionen später zu überlagern. Viele Implementierungen versuchen deshalb, mit einem Daten- und Textsegment nach `fork` auszukommen, bis ein `exec` Aufruf beim Sohn erfolgt.

Am vereinfachten Beispiel des Systemstarts können Sie in der Abbildung unten nochmals das Zusammenspiel von fork und exec — abgekürzt durch (*) — beobachten. Für den



zweiten Bildschirm ist der Anmeldevorgang des Teilnehmers fix angedeutet. Nachdem getty von login überlagert wurde, wird dieses Bild erneut beim erfolgreichen Anmelden von der Benutzershell, also z. B. /bin/sh, überschrieben, jeweils innerhalb desselben weiterbestehenden Prozesses und damit mit gleicher Prozeßnummer.

Nach dem Abmelden stirbt dieser Prozeß aber dann mit der letzten Shell und init wird einen neuen getty-Prozeß für das Terminal erzeugen, der auf Anmeldungen wartet. Die tatsächlichen Abläufe gegenüber den hier angedeuteten sind allerdings heute durch die Vernetzung der Rechenanlagen etwas komplizierter geworden, d. h. es kommt der Aufbau der Netzverbindung nach außen und das Laden (mount) verteilter Dateisysteme im lokalen Netz dazu.

Eine weitere Schwäche von UNIX ist, daß zwischen Vater und Sohn wenig Synchronisation und Kommunikation möglich ist. Nach dem Aufruf von fork kann der Vater nur mit wait auf die Beendigung des Sohnes warten. Dieser beendet seine Arbeit, wenn er am Ende seines Programms angekommen ist, oder durch exit. Sowohl wait als auch exit sind Shell-Spezialkommandos und benutzen die gleichnamigen Systemaufrufe.

Für die Kommunikation zwischen dem Betriebssystemkern und Prozessen (z. B. bei Stapelüberlauf), zwischen Prozessen untereinander und zwischen Benutzer und Prozeß (z. B. Interrupt-Taste drücken) gibt es eine bescheidene Anzahl von *Signalen* (üblicherweise eine ganze Zahl zwischen 1 und 19). Neuere Versionen haben hier seit langem mehr Komfort gebracht, z. B. *sockets*, *streams* und *semaphores* (siehe Einschub unten).

Die elementare Signalverarbeitung erfolgt über die vier Systemaufrufe:

`pause( )`	Warten auf ein Signal
`signal( )`	Programmstück für Ausführung bei Eintreffen des Signals
`alarm( )`	Von der Zeitverwaltung ein Signal nach der angegebenen Zeit bestellen
`kill( )`	Senden eines Signals

Den Systemaufruf `kill` gibt es auch als Kommando. Mit `kill -9` (Signal `9`) läßt sich jeder Prozeß umbringen, da es nicht abgefangen oder ignoriert werden kann. Man sollte es nur in Notfällen benutzen, da Aufräumarbeiten entfallen. Das Standardsignal ist `15`. Natürlich kann man nur Prozesse beenden, die einem selbst gehören, es sei denn, man ist `root`.

Ein kritischer Abschnitt

UNIX war von Anfang an ein Mehrbenutzerbetriebssystem und hat schon früh als ideale Plattform für Entwicklung und Einsatz von Datenbanken gegolten, bei denen bekanntlich parallel ablaufende Transaktionen eine große Rolle spielen. Dies muß eigentlich überraschen, denn UNIX ist fast ausschließlich in C implementiert und C kennt, anders als z. B. die Programmiersprachen Modula oder Ada, keine Sprachmittel für die Prozeßsynchronisierung.

Bereiche, in denen zu jedem Zeitpunkt höchstens ein Prozeß aktiv sein kann, nennt man kritische Abschnitte, die man sich wie eine eingleisige Strecke in einem Eisenbahnnetz mit vielen Zügen vorstellen kann. In einem Rechner können dies Ein-/Ausgabepuffer, Dateien, Sätze oder Systemtabelleneinträge sein. Aufgabe der *Prozeßsynchronisierung* ist es, den Zutritt zu diesen kritischen Abschnitten zu regeln.

Von der Hardware wird diese Aufgabe dadurch unterstützt, daß es mindestens eine Instruktion gibt, die atomar, d. h. untrennbar ohne Erlaubnis zur Prozeßumschaltung oder Unterbrechung, einen Wert liest und ihn danach setzt. Wohlgemerkt, dieses atomare Verhalten gilt für eine Maschineninstruktion und eine Hauptspeicherzelle, also heutzutage für weniger als eine Hundertstel Microsekunde ($< 10^{-8}$ s), keineswegs läßt sich damit etwa ein Dateizugriff atomar abwickeln, der praktisch beliebig lange dauern kann!

Der Ausweg besteht jetzt darin, mit der kurzen, atomaren Maschineninstruktion sichere Softwareschalter zu bauen, mit denen man komfortabel kritische Abschnitte schützen kann, auch wenn während des Aufenthalts im kritischen Abschnitt mehrmals der Prozessor von Prozeß zu Prozeß wechselt[1]. Wegen der gerade erwähnten Schwächen von C in der Prozeßsynchronisierung waren, wie Rochkind anschaulich in [13] schildert, in den Anfangsjahren von UNIX die Softwareschalter leere Dateien auf deren Namen sich die beteiligten Prozesse geeinigt hatten! Dies funktionierte, weil das Anlegen einer nicht existierenden Datei mit `creat()` unabhängig von den Schreibrechten *immer funktionierte*,

1. Dies ist kein Widerspruch: Zwar kann durch einen Prozeßwechsel ein anderer Prozeß versuchen, den kritischen Abschnitt zu betreten, es wird ihm aber aufgrund des Softwareschalters nicht gelingen, so daß er den Prozessor wieder abgeben und auf die Freigabe warten wird (vgl. Abschnitt 3.3).

das Leeren einer existierenden Datei mit `creat()` bei normalen Benutzern (nicht dem super-user) *nicht funktionierte,* wenn diese schreibgeschützt war.

Kritische Abschnitte können demnach zuerst und exklusiv von dem belegt werden, dem es gelingt, die vereinbarte, zugehörige leere Datei anzulegen. Ab System III kamen die bereits in Abschnitt 3.2 erwähnten FIFOs (named pipes) dazu, die Prozeßsynchronisierung **und** Prozeßkommunikation bieten. Sind FIFOs recht aufwendige — und damit langsame — Synchronisierungswerkzeuge, so wird spätestens seit System V niemand mehr mit „Tricks“ à la `creat()` arbeiten, sondern mit den dafür vorgesehenen, schnellen Semaphoren, einer von Dijkstra bereits 1965 vorgeschlagenen Methode mit einer P- und einer V-Operation für das Belegen und Freigeben des Softwareschalters. Der Begriff Semaphor stammt übrigens aus der Schiffahrts- und Eisenbahnersprache und bezeichnet ein von beiden Einfahrten sichbares Wendesignal für einen einspurigen Abschnitt[1].

Die Semaphor-Implementierung in UNIX, speziell auch gedacht für den Zugriff auf gemeinsame Speicherbereiche (shared memory), verwendet die Systemaufrufe `semget`, `semctl`und `semop` und ist insgesamt recht undurchsichtig. Zur Verteidigung muß man aber anführen, daß die Implementierung auch den in der Praxis durchaus auftretenden Fall eines „Prozeßabsturzes“ im kritischen Abschnitt berücksichtigt, den man in der Theorie gerne ausschließt und der ohne Behandlung leicht zu einer gesperrten Platte und anderen Hängepartien führen kann.

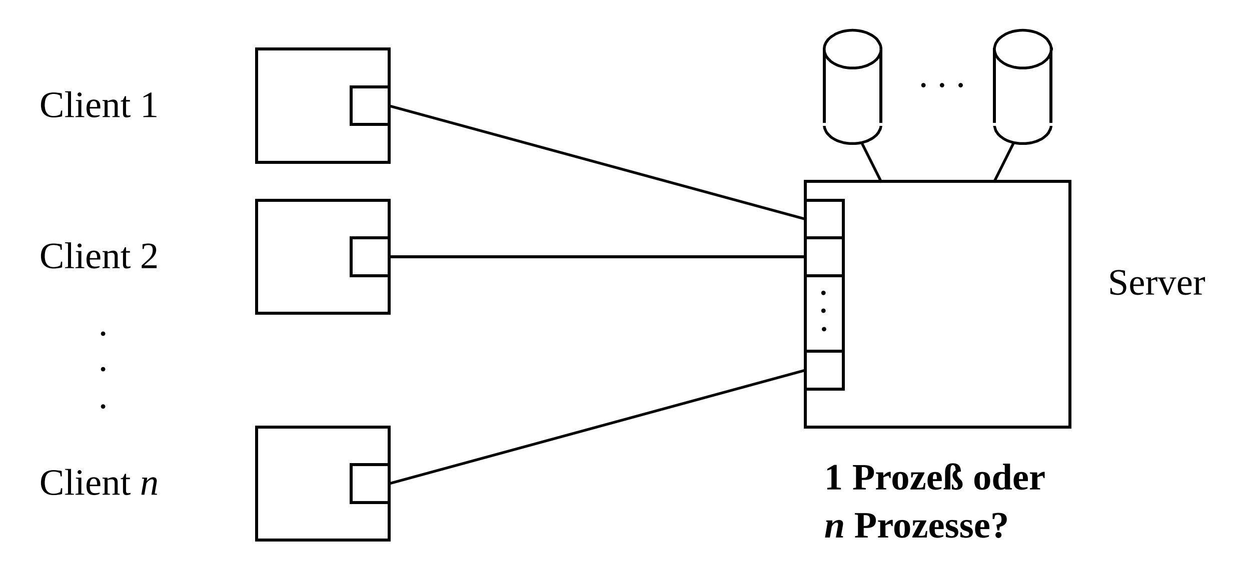
Semaphore sind aber von Natur aus unhandlich. Deshalb hat man schon früh nach anderen Synchronisationsmethoden geforscht. Die bekanntesten Konzepte sind

- *Monitore* (Brinch-Hansen, Hoare, ca. 1973/74, vgl. [2, 14])
- *Remote Procedure Call* (rpc) zur Realisierung eines üblicherweise synchronen (Absender wartet auf Antwort) Botschaftenkonzepts; orientiert sich an der Ähnlichkeit mit Prozeduraufrufen (Aufrufer wartet bis Aufruf abgearbeitet ist und die Routine zurückkehrt); setzt in der UNIX-Variante ein gemeinsames Übertragungsprotokoll voraus, arbeitet mit einem generischen (immer passenden) Aufruf der die Wandlung aller Argumente in ein architektur-neutrales Format beinhaltet und erlaubt als Preis für all’ die Mühen die Synchronisierung über unterschiedliche Rechnerstrukturen hinweg,
- *Streams* (Ritchie 1984, vgl. [12]) als UNIX-Realisierung ebenfalls eines Botschaftenkonzepts, besonders für die zeichenorientierte, bi-direktionale Verbindung eines Gerätetreibers im Kernel mit einem Anwenderprozeß,
- *Sockets* als recht komfortable Schnittstelle eines Anwendungsprogramms zur Kommunikation mittels TCP/IP (Transport Connect Protocol/Internet-Protokoll), d.h. die Aufrufe der Sende- und Empfangsroutinen ähneln stark denen der Ein-/Ausgabe, zusätzlich kommen noch spezielle Funktionen für Verbindungsaufbau, Namenserfragung, usw. dazu.

1. Verzeihung, die Anspielung drängt sich zu sehr auf: Dieses Kursbuch stellt nicht nur die Weichen, es setzt auch Signale.

Zuletzt muß im Zusammenhang mit der oben erwähnten Datenbankanwendung, die nach dem *Client-Server-Modell* aufgebaut sein kann, noch ein anderes Problem angesprochen werden:

- Ist der Server als ein Prozeß implementiert, kann der Seitenfehler eines einzigen Clients den Server für die Dauer des Lesevorgangs blockieren.
- Ist der Server als Sammlung von *n* Prozessen implementiert, ist es schwierig, den Seitenpuffer für die optimale Zugriffssteuerung, die Log-Datei für das Zurücksetzen von Transaktionen, die globale Sperrtabelle und viele andere Ressourcen gemeinsam zu nutzen.



UNIX bietet als Ausweg die sog. *Threads* (*engl*. Fäden) an. Mit diesem Packet, das in der *Distributed Computing Environment* (DCE) Ausführung als Programmbibliothek und nicht als Kernerweiterung ausgeführt ist, kann der Anwender seine eigenen Pseudoprozesse steuern und synchronisieren und regelt für sie die Prozessorvergabe. Weil die aufwendige Kontextumschaltung zwischen echten Prozessen wegfällt, spricht man auch von sog. *leichtgewichtigen Prozessen*. Threads kommunizieren über gemeinsame Variablen miteinander und können mit verschiedenen Methoden, darunter den Hoare'schen Monitoren, synchronisiert werden.

Allerdings geht mit dieser Lösung die gesamte Verantwortung auf den Anwender, also in unserem Beispiel das Datenbankentwicklerteam, über. Sind externe Signale berücksichtigt, Prioritäten eingearbeitet, Ausnahmebehandlungen vorgesehen? Werden Verklemmungen entdeckt, Zeitüberschreitungen festgestellt? Die Header für Threads sind auf SUN-Rechnern in `lwp.h` (*light weight processes*), für AIX in `pthread.h` zu finden: Falls Sie die nächsten fünf Jahre nichts dringendes vorhaben, wir bräuchten für unseren Datenbankprototypen auch noch eine Transaktionskomponente!

Einblick in die Internas von UNIX geben sowohl Bach [1] als auch Stevens [15], letzterer mehr mit Blick auf den Systemprogrammierer. Warum es in UNIX mehr als zehn Prozeßkommunikationsmöglichkeiten gibt, von denen keine alle Ansprüche befriedigt, schildert

kurz und gut lesbar Rochkind [13]. Die immer noch beste Behandlung der Prozeßsynchronisierung aus Programmiersprachensicht ist Ben-Ari [2], ein weitverbreiteter Standardtext zu Betriebssystemen stammt von Silberschatz und Galvin [14].

Die letzte Schwäche, von der hier die Rede sein soll, betrifft das Abschalten von UNIX. Auch wenn man alleine am Rechner sitzt, wird es einem nicht gelingen, den Prozeß `init` „umzubringen“. Das deshalb öfters praktizierte „Steckerziehen“ hat unter Umständen üble Folgen, weil das Dateisystem in einem inkonsistenten Zustand hinterlassen werden kann. Sofern es für den letzten Benutzer keine Routine zum Herunterfahren gibt, sollte man es dem super-user überlassen, mit dem Kommando `shutdown` die Anlage abzuschalten.

```
$Broadcast Message from root
Shutdown in 1 minute.
Clean up and log off.

System going down in 60 seconds
```

So wie es Herrn Professor Fix jetzt passiert. Zeit den Kurs zu beenden.

Vieles konnte nicht behandelt werden. Einige Feinheiten der Shell, `$@` und die Fluchtsymbole, haben wir ausgelassen. Zur effektiven Nutzung wird man sich ferner die Mustersetzungssprache `awk`, den Stream Editor `sed` und alle Optionen des Kommandos `grep` anschauen müssen.

```
System going down in 30 seconds
Please log off.
```

Ernsthafte UNIX-Programmierer werden die Systemaufrufe aus dem zweiten Teil des UNIX Manuals vermissen. Mehr an den Anwendungen Interessierte sollten einen Blick in die wohlbestückte Werkzeugkiste mit Terminalanpassung, `yacc`, `lex` für den Compilerbau und `make`, `sccs` zur Softwareerzeugung werfen. Auch die Pakete zur Druckaufbereitung `nroff`, `troff`, `eqn`, `tbl` und `ms` gehören zu UNIX.

```
System shutdown time has arrived
Fri Apr 4 16:10 1997
** Normal System Shutdown **
```

Das war es dann wohl. Viel Spaß mit UNIX und danke für Ihre Geduld.

Zusammenfassung

❖ Wir lernten die `fork-` und `exec` Systemaufrufe, sowie mehr über Aufbau, Verwaltung und Rolle der *Prozesse* in UNIX kennen.

Frage 10

In der Texttabelle für die Verwaltung der Instruktionssegmente gibt es einen Zähler für die Anzahl der Benutzer. Welche Aussage ist für gleichzeitig nutzbare (read-only) Segmente richtig?

❑ Solange der Zähler größer als null ist, kann das Segment nicht verdrängt werden.

❑ Wenn der Zähler null ist, wird das Segment auf die Platte zurückgeschrieben.

❑ Wenn der Zähler null ist, kann das Segment zum Überschreiben freigegeben werden.

Frage 11

Zur Abarbeitung des Shell-Skripts `lsr`

❑ überlagert sich die interaktive Shell mit einer Untershell, die `lsr` abarbeitet.

❑ erzeugt die interaktive Shell eine Untershell, die sich mit `lsr` überlagert.

❑ erzeugt die interaktive Shell eine Untershell, die `lsr` abarbeitet.

Frage 12

Können die Klammern wegfallen?

```
export leer
for i
do
    if test
    then ( echo "$leer$i:" ; cd $i
        leer=$leer"  " ; lsr `ls` )
    else echo "$leer$i"
    fi
done
```

?

❑ Ja.

❑ Nein, sie sind syntaktisch (if-then-else-fi) notwendig.

❑ Nein, sie sind semantisch (Untershell) notwendig.

Frage 13

Die primitive Methode, mit zweimal `nn` die Anzahl der durch `lsr` verbrauchten Prozesse zu zählen, liefert unterschiedliche Werte für denselben Dateibaum. Der Grund ist

❑ die Unzuverlässigkeit von `nn`.

❑ der Mehrbenutzerbetrieb.

❑ die jeweilige Tagesform der Anlage.

- Sie haben jetzt das Ende der zehnten Lektion erreicht und damit das Ende des UNIX Kursbuchs. Wir wünschen Ihnen viel Erfolg bei der Arbeit mit dem UNIX Betriebssystem.

Anhang: Lösungen zu den Fragen des Kurses

1. Einführung

Frage 1 Die Behauptung ist falsch. UNIX ist fast durchweg in der höheren Programmiersprache C geschrieben, nur ganz geringe Anteile im Assembler der jeweiligen Maschine.

Frage 2 Die Behauptung ist falsch. UNIX entstand ab 1969 aus Eigeninitiative von Ken Thompson und Dennis Ritchie. Erst später wurde es zu einem kommerziellen Produkt.

Frage 3 Die erste Antwort ist richtig. Die meisten Systeme basieren auf einer Lizenz von AT&T Bell Laboratories, sind POSIX-kompatibel und bieten bei unverändertem Kernsystem nur neue interaktive Arbeitsumgebungen an.

Frage 4 Die dritte Antwort ist richtig. Der *login-Name* (maximal 8 Zeichen, keine Großbuchstaben) wird vom Benutzer frei gewählt und vom Systemverwalter eingetragen. Im Gegensatz zum (verschlüsselten) Paßwort ist er von allen Benutzern einsehbar, da hierüber auch Botschaften (`mail`) adressiert werden.
Neben dem login-Namen wird noch eine Benutzernummer und eine Gruppennummer vergeben (siehe Abschnitt 7.1).

Frage 5 Der Systemverwalter heißt *super-user*. Wenn Sie es wußten: Vielleicht haben Sie ja selbst das Zeug dazu! Der super-user trägt neue Benutzer im System ein und kann als einziger die Paßwörter anderer Teilnehmer ändern.

Frage 6 Die Tastenkombination zum Abmelden ist ’*CTRL-d*’. Sie signalisiert auch dem Kommandointerpreter das Ende der Sitzung. Die Kombination ’*CTRL-s*’ hält das Bildschirmscrollen an, ’*CTRL-q*’ setzt die Bildschirmausgabe fort und ’*CTRL-c*’ dient meist zur Programmunterbrechung

Frage 7 Das Kommando hat die vier Argumente „`Hallo`“, „`Ihr`“, „`da`“ und „`!`“, die jeweils durch ein Leerzeichen voneinander getrennt sind.

Frage 8 Die erste Antwort ist richtig. S.R.Bourne, nach dem diese Standard-Shell benannt ist, ist ein Mitautor des UNIX-Systems.

Frage 9 Das dritte Kommando könnte zu einem Fehler führen, da mit `passwd echo` nur der Benutzer mit dem login-Namen `echo` sein eigenes Paßwort ändern kann, wobei das Argument `echo` dann überflüssig ist, oder der super-user das Paßwort für den Benutzer `echo`, sofern ein solcher eingetragen ist.

2. Ein- und Ausgabeumlenkung

Frage 1 Die richtige Antwort lautet `tty`. Diese Abkürzung taucht in UNIX häufig auf, z.B. in

```
$who am I
fix tty7 Dec 15 11:22
$
```

`tty` steht für *teletype*, eine veraltete Bezeichnung für Terminals aus der Fernschreiberzeit.

Frage 2 Die erste Antwort ist richtig. Professor Fix erhält beim nächsten login einen Bief von sich selbst, z.B. das Memo : Nicht vergessen, Angebot für UNIX-Rechner einholen!

Frage 3 Die zweite Antwort ist richtig. '*CTRL-d*' signalisiert einem lesenden Programm das Eingabeende. An `mail` geschickt, bewirkt es die Beendigung von `mail` und die Rückkehr zur Shell. An die Shell geschickt, bewirkt es die Beendigung der Shell und damit die Rückkehr zur Terminalinitialisierung, d.h. die Abmeldung.

Frage 4 Mit dem Editorkommando `m neu` (`m` steht für **m**ail on, weitersenden). In komfortableren Mail-Editoren heißt das Kürzel meist `f` (für **f**orward, weiterreichen).

Frage 5 Die erste Antwort ist richtig. Die Datei `zaehlen` enthält 56 sichtbare Zeichen und 3 NL-Zeichen, am Ende jeder Zeile eines.

Frage 6 Antwort zwei ist richtig. Dann mal ab die Post!

Frage 7 Die Shell ruft das Kommando `cat` mit dem Argument `gruss` auf. Das führt zu der Ausgabe `Hallo Leute!`, wie aus der ersten Zeile des Dialogs zu ersehen ist.

Frage 8 Die dritte Antwort ist richtig.
Bei der ersten Alternative werden die Inhalte der Dateien `raum`, `gesehen` und `date`, sofern vorhanden, an die Datei `raum` angefügt.
Bei der zweiten Variante werden die beiden Worte `gesehen` und `date` an die Datei `raum` angefügt.
Der vierte Vorschlag fügt das Wort `gesehen` und den Inhalt der Datei `date`, sofern vorhanden, an die Datei `raum` an.

Frage 9 Zeile drei kann nicht gut gehen, sofern `Hallo` nicht als Kommando existiert.

3. Kommandos im Zusammenspiel

Frage 1 Die richtige Option wäre `s` für **s**queeze (zusammenschieben).

Frage 2 Die erste, dritte und vierte Alternative sind gleichwertig. `lpr -r n ankuend` hingegen wäre die Aufforderung, die Dateien `n` und `ankuend` auszudrucken — syntaktisch korrekt, aber nicht gemeint.

Frage 3 Die zweite Antwort ist richtig.
Die Verwendung von `datei >lpr` zum Ausdrucken einer Datei ist ein häufiger Flüchtigkeitsfehler. Richtig wäre dafür `lpr <datei` oder `lpr datei` oder `cat datei | lpr`.

Frage 4 Die Antwort ist die Ausgabe `1` durch die Pipeline. Das erste `wc` Kommando liefert **eine** Zeile mit drei Zahlen (Zeilen, Wörter, Zeichen). Da das zweite `wc` Kommando nur die Zeilen zählt (Option `-l`), ist das Ergebnis 1.

Frage 5 Die erste Aussage ist falsch, denn ein wartender Prozeß kann nie direkt die CPU zugeteilt bekommen, sein Zustand muß sich erst in „bereit“ ändern.

Frage 6 Die erste Antwort ist richtig, denn Mehrbenutzerbetrieb bedeutet mindestens einen Prozeß je Teilnehmer und damit Mehrprogrammfähigkeit.

Frage 7 Die Aussage ist falsch, denn eine der Stärken von UNIX ist gerade das einheitliche Prozeßkonzept für System- und Anwenderaufgaben.

Frage 8 Die zweite Anwort ist richtig, denn wie wir gesehen haben, wurde `ps` als rechnender Prozeß (Status `r`) ausgegeben.

Frage 9 Die zweite Antwort ist richtig. Generell kann man so durch Einbau eines `at` Kommandos im Shell-Skript einen Batch-Job erneut ansetzen.

4. Das hierarchische Dateisystem

Frage 1 Der volle Pfadname lautet `/usr/fix/raum`.

Frage 2 Der relative Pfadname ist `raum`.

Frage 3 Ja, beide Kommandos liefern die gleiche Ausgabe.

Frage 4 Die zweite Antwort ist richtig, denn die Datei `tricky` wird zuerst von der Shell angelegt und dann von `ls` bereits mit ausgegeben.

Frage 5 Die dritte Antwort ist richtig. Wäre bei `cd` des Arbeitsverzeichnis gemeint, bliebe der Zeiger stehen, wo er ist. Wäre bei `ls` das Heimatverzeichnis gemeint, würden immer die Einträge des selben Verzeichnisses ausgegeben.

Frage 6 Die dritte Antwort ist richtig. Die erste Antwort kann mit der Option `t` erzwungen werden, die zweite mit der Option `nt`.

Frage 7 Die zweite Antwort ist richtig. Die datei `lekt3` enthält nur ein NL-Zeichen, das ein **Begrenzer** für Wörter und natürlich für Zeilen ist. Die Ausgabe lautet deshalb 1 0 1, d.h. eine Zeile, null Wörter und ein Zeichen.

Frage 8 Die erste und dritte Antwort sind richtig, wobei wir voraussetzen, daß das Verzeichnis `usr` wie bisher direkt unter der Wurzel liegt.

Frage 9 Das zweite Kommando ist sinnlos, denn es wäre ein Wechsel zum selben Ort.

Frage 10 Hier sollten Sie sechs Richtige haben! Die Dateien `test/x/a/b`, `test/x/a/c`, `test/x/a`, `test/x/d`, `test/x` und `test/z` werden gelöscht.

Frage 11 Die dritte Antwort ist richtig.

5. Dateien kopieren und verlagern

Frage 1 Die zweite Antwort ist richtig. Die beiden Kommandos liefern nur dann die gleiche Ausgabe, wenn die Dateien `bs` und `do` gleich sind. Bei Vertauschung der Argumente wird z.B. eine vorher anzufügende Zeile (`a`) als zu löschende Zeile (`d`) markiert.

Frage 2 Die zweite Anwort ist richtig, denn `1,$` steht für den ganzen Puffer des Editors und `g` (global) bedeutet Änderung an allen Stellen.

Frage 3 Die zweite Alternative versagt. Es führen zwar viele Wege nach Rom, aber dieser ist der Holzweg, weil die Dateien `bs` und `do` nach der Änderung des Arbeitskataloges so nicht mehr anzusprechen sind.

Frage 4 Durch das move-Kommando, also mit `mv do bs aushang`.

Frage 5 Die erste Antwort ist richtig, denn ein Dateibaum kann nicht in einen eigenen Teilbaum verlagert werden.

Frage 6 Die dritte Antwort ist richtig.

Frage 7 Das fehlende Kommando lautet `write fix`. Zum Testen von `write` kann man durchaus Selbstgespräche führen.

Frage 8 Die dritte Antwort ist richtig. Beachten Sie, daß nach `cd umlauf` für das link-Kommando der Name `umlauf` kein Katalog ist, sondern der Name für eine Normaldatei. Deshalb wird die Datei `/usr/fbinfo/ausflug` unter dem weiteren Namen `umlauf` im Arbeitsverzeichnis `/usr/fix/umlauf` eingetragen.

Frage 9 Es sind zwölf Einträge, der Verweis des Vorgängers, der Verweis auf sich selbst (Punkt) und zehnmal der Rückwärtsverweis (PunktPunkt) der Unterverzeichnisse. Wir geben aber zu, daß diese Frage besonders schwierig war.

Frage 10 Die erste Antwort ist richtig. Alle Editoren arbeiten auf einer Kopie der Datei, falls Änderungen nachträglich widerrufen werden.

6. Schutzmechanismen

Frage 1 Die dritte Antwort ist richtig. Wenn ein Verzeichnis schreibgeschützt ist, kann es nicht geleert werden und damit auch nicht mit `rmdir` aus dem Vorgängerkatalog entfernt werden.

Frage 2 Die dritte Antwort ist richtig.

Frage 3 Die Antwort lautet `755`. Das ergibt sich aus der üblichen Schreibsperre für Gruppe und Welt: `rwxr-xr-x`, damit dual `111101101` oder oktal `755`.

Frage 4 Die dritte Antwort ist richtig. Der super-user mußte eingreifen. Heutzutage landet der Benutzer meist in „`/`“ und kann von dort sein x-Recht wieder setzen.

Frage 5 Die zweite Antwort ist richtig, denn `$1` steht für die neu angelegte Datei, in unserem Fall `prog`.

Frage 6 Die zweite Antwort ist richtig, also Vorsicht.

Frage 7 Die zweite Antwort ist richtig. Das macht zwar einige Arbeit, ist aber nicht so schwierig. Die dritte Antwort ist falsch, denn der Schlüssel kommt von `/dev/tty`, andernfalls wäre die erste Zeile des Klartextes der Schlüssel.

Frage 8 Die vierte Antwort ist richtig, denn Sie verlieren die Datei bei einer falschen Eingabe.

Frage 9 Die zweite Antwort ist richtig. Sie selbst können jedenfalls nichts mehr reparie-

ren und am Benzin liegt's auch nicht (Alternative 4).

7. Besitzverhältnisse

Frage 1 Die vierte Antwort ist richtig. Alternativ zu `id` könnte man z.B. mit `mkdir t; ls -lg; rmdir t` die Gruppenkennung erfahren.

Frage 2 Die Antworten drei und vier sind falsch, denn eine Datei hat genau eine Gruppe und einen Teilnehmer als Besitzer.

Frage 3 Die erste Antwort ist richtig, denn das Programm zum Laden der Tabelle kann ja die Berechtigung des Aufrufers prüfen.

Frage 4 Die dritte Antwort ist richtig. Ein Leseschutz bringt keine zusätzliche Sicherheit, und alle sollen das Programm aufrufen können.

Frage 5 Die zweite Antwort ist richtig, denn nur das Programm muß als effektiven Besitzer die Kennung `root` haben.

Frage 6 Die zweite Antwort ist richtig. Das löschen eines Katalogeintrages verlangt die effektive Kennung `0` von `root`.

Frage 7 Die erste Antwort ist richtig. Es handelt sich um eine Sicherheitslücke aus den frühen UNIX-Tagen. Sie wurde beseitigt durch automatisches Löschen des suid-Bits bei Ausführung des Kommandos `chown`.

8. Benutzung der Shell

Frage 1 Die richtige Antwort lautet `0`. Zuerst wird `test` als leere Datei eröffnet, danach `ls -l` ausgeführt, dann erst wird in `test` geschrieben.

Frage 2 Die erste Antwort ist richtig, im Verlauf des Kurses wird auch gezeigt, wie man die Änderungen zur nächsten Sitzung rettet.

Frage 3 Die dritte Antwort ist richtig und damit ist die Zeile wenig zu empfehlen.

Frage 4 Die richtige Antwort lautet `6`. Das Kommando `ls` wird mit den Argumenten Punkt (`.`) und PunktPunkt (`..`) aufgerufen, liefert also `bs`, `do`, `aushang`, `nachb3`, `raum` und `studord`.

Frage 5 Die richtige Antwort lautet `2`. Das Kommando `echo` erhält als Argumente die Dateinamen `aushang` und `raum`, die mit `a`, bzw. `r` aus `[amru]` beginnen.

Frage 6 Die zweite Antwort ist richtig. Das Fluchtsymbol \ hebt die Wirkung des folgenden Zeichens auf, auch in `"..."`, aber nicht in `'...'`.

Frage 7 Die richtige Antwort lautet: `postda jr Paket`. Nicht richtig wäre `postda Paket jr`.

Frage 8 Die dritte Antwort ist richtig. Hier erscheint bei Mißerfolg durch `who` die Liste der Teilnehmer.

Frage 9 Die zweite Antwort ist richtig. Durch `1>&2` wird die Standardausgabe von `wc` in die Standardfehlerausgabe umgeleitet.

9. Die Parameter der Shell

Frage 1 Die zweite Antwort ist richtig. `lfrage` ist das erste Wort und damit `$0`.

Frage 2 Die dritte Antwort ist richtig. Überschüssige Argumente werden ignoriert, eine gefährliche Angelegenheit.

Frage 3 Die zweite Antwort ist richtig. Die Erzeugung neuer Untershells kann beliebig oft erfolgen, ist aber aufwendig in der Abarbeitung.

Frage 4 Die erste Antwort ist richtig. `"Anruf auf ... Dekan"` ersetzt `$1`, `fix` ersetzt `$2`.

Frage 5 Die zweite Antwort ist richtig - ziemlich einfach, oder?

Frage 6 Die zweite Antwort ist richtig. Gegebenenfalls werden nicht vorhandene Argumente als leere Zeichenketten ausgegeben.

Frage 7 Die erste Antwort ist richtig. `cd` ohne Argumente wechselt ins Heimat-verzeichnis.

Frage 8 Die erste Antwort ist richtig.

10. Die Shell und ihre Umgebung

Frage 1 Die erste und die dritte Antwort sind richtig. `cat verteiler` setzt voraus, daß `verteiler` weiterhin besteht und im aktuellen Katalog abgelegt ist. Auch `echo` ... ist eine praktikable Lösung, da die Adressaten in `vt` abgelegt sind.

Frage 2 Die richtige Antwort lautet: `&&`. Wenn Sie die richtige Antwort wußten, dann

schnell `exit 0` und weiter zur nächsten Frage.

Frage 3 Die richtige Antwort lautet: `y`. `mesg y` für **y**es gibt das Terminal wieder frei, `mesg` ohne Argument zeigt die Empfangsbereitschaft (ein/aus) an.

Frage 4 Die dritte Antwort ist richtig.

Frage 5 Die zweite Antwort ist richtig. Ein Punkt für Sie?

Frage 6 Die erste Antwort ist richtig, denn sonst braucht man keinen Namen mit Wert zu übergeben.

Frage 7 Die zweite Antwort ist richtig. Alle Wörter werden `empfaenger` zugewiesen und damit in `mail` eingesetzt.

Frage 8 Die zweite Antwort ist richtig. `echo` gibt die Kontrolle an `init` zurück, so wie es die Shell bei EOT macht.

Frage 9 Die dritte Antwort ist richtig. Die Zuweisungen würden dann in der aktuellen Shell vorgenommen.

Frage 10 Die dritte Antwort ist richtig. Das Segment muß wegen `read-only` nicht zurückgeschrieben werden.

Frage 11 Die dritte Antwort ist korrekt. Lagen Sie richtig?

Frage 12 Die dritte Antwort ist richtig. Nach Rückkehr aus der Untershell ist die betrachtete Wurzel wieder aktueller Katalog, analog einer Rekursion mit Werteparameter-Übergabe.

Frage 13 Die zweite Antwort ist richtig. Prozeßwechsel geschehen quasi zufällig.

Literatur

[1] Maurice J. Bach, The Design of the UNIX Operating System, Prentice-Hall, Englewood Cliffs, N.J., 1986, deutsch unter dem Titel „UNIX - Wie funktioniert das Betriebssystem?“, Hanser, München, 1991

[2] M. Ben-Ari, Grundlagen der Parallel-Programmierung, Hanser, München, 1984

[3] D. Cameron and B. Rosenblatt, Learning GNU Emacs. O’Reilly, Sebastopol, CA, 2nd ed., 1996

[4] Simson Garfinkel,: PGP: Pretty Good Privacy - Verschlüsselung von E-Mail, O’Reilly, Sebastopol, CA, 1996

[5] Wilfried Grieger, Wer hat Angst vorm Emacs?, Addison-Wesley, Bonn, 1994

[6] J. Gulbins und K. Obermayer, UNIX System V.4. Eine Einführung in Begriffe und Kommandos, Springer Compass, Berlin Heidelberg, 4. Auflage 1995

[7] B. W. Kernighan and R. Pike, The UNIX Programming Environment, Prentice-Hall, Englewood Cliffs, N.J., 1984, deutsch unter dem Titel „Der UNIX-Werkzeugkasten“, Hanser, München, 1987

[8] Arnold Klingert, Einführung in Graphische Fenstersysteme, Springer, Berlin Heidelberg, 1996

[9] John K. Ousterhout, Tcl und Tk, Addison-Wesley, Bonn, 1995

[10] D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, Comm. ACM, Vol. 17, No. 7, July 1974, pp. 365-375; nachgedruckt im Jubiläumsheft der Comm. ACM, Vol. 26, No. 1, Jan. 1983; die Turing Award Vorträge finden sich in Comm. ACM, Vol. 27, No. 8, Aug. 1984.

[11] D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, AT&T The Bell System Tech. J., Vol. 57, No. 6, July-August 1978, reprinted in : UNIX SYSTEM Readings and Applications, Prentice-Hall, Englewood Cliffs, N.J., Vol. 1, pp. 1-24, 1987
(im ersten der beiden von den Bell Labs publizierten Sonderhefte zu UNIX, die beide 1987 nachgedruckt wurden)

[12] D. M. Ritchie, A Stream Input Output System, AT&T The Bell Laboratories Tech. J., Vol. 63, No. 8, October 1984, reprinted in : UNIX SYSTEM Readings and Applications, Prentice-Hall, Englewood Cliffs, N.J., Vol. 2, pp. 311-324, 1987
(im zweiten der beiden von den Bell Labs publizierten Sonderhefte zu UNIX, die beide 1987 nachgedruckt wurden)

[13] M. J. Rochkind, Advanced UNIX Programming, Prentice-Hall, Englewood Cliffs, N.J., 1985

[14] A. Silberschatz and P. B. Galvin, Operating System Concepts, Addison-Wesley, Reading, Mass., 4rd Ed. 1994

[15] W. Richard Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley, Reading, Mass., 1992

[16] Stefan Strobel und Thomas Uhl, LINUX - Vom PC zur Workstation, Springer, Berlin Heidelberg, 1994

Index

C

D

E

F

G

H

Q

R

S

T

U

V